Acore: The Design of a Core Actor Language and its Compiler

by

Carl Roger Manning

S.B., Computer Science
Massachusetts Institute of Technology
(1985)

**Submitted in partial fulfillment
of the requirements for the
degree of**

Master of Science

at the

**Massachusetts Institute of Technology
May, 1987**

Signature of Author .....
Department of Electrical Engineering and Computer Science
May 1987

Certified by .....
Professor Carl E. Hewitt
Thesis Supervisor

Accepted by .....
Professor Arthur C. Smith
Chairman, Committee on Graduate Students

Acore: The Design of a Core Actor Language and its Compiler

by

Carl Roger Manning

Submitted to the
Department of Electrical Engineering and Computer Science
May, 1987 in partial fulfillment of the requirements
for the degree of Master of Science

Abstract

As VLSI brings the prospect of general purpose parallel computer architectures closer to reality, people are interested in developing software systems which exploit that concurrency. To define and experiment with these software systems, concurrent programming languages are needed; in particular, to be able to express and use the full power of concurrent processing, we need languages which are capable of expressing diverse kinds of concurrent processes, including processes with cooperating, competing, and resource-limited sub- or co- processes. Language design can proceed in many different directions; I present the design and implementation a *core* actor language which will serve as a basis for further development of concurrent languages.

Thesis Supervisor: Professor Carl E. Hewitt

Acknowledgments

This thesis builds on the work of many members, past and present, of the Message Passing Semantics Group at the MIT Artificial Intelligence Laboratory. Some I have known, but many I have never met, so I won't try to list them all here.

I received much helpful feedback from comments and discussions with current members of the group, especially Tom Reinhardt and Carl Hewitt. Gul Agha was always ready to answer questions about details of his formulation of the actor model.

I thank everyone, but I must especially thank my thesis advisor, Carl Hewitt, who spent many hours working his way through rough drafts of this thesis at the last moment.

Carl Manning

Table of Contents

Table of Figures

Part I
Introduction

Chapter One

Introduction

As VLSI brings the prospect of general purpose parallel computer architectures closer to reality, there is a growing interest in building concurrent software systems which can take advantage of that parallelism. This is especially true in fields such as artificial intelligence, where the additional power provided by concurrency is seen as one key needed to open up new ideas and possibilities for research and experimentation. The new power is not just the power of more speed; the additional power of concurrent software systems comes from the ability to construct organizations of evolving concurrent processes, rather than being limited to building a sequential process operating over an organization of evolving objects. To open up the possibilities of concurrency for experimentation, languages are needed which can express diverse kinds of concurrent processes. Current designs for concurrent languages are restricted by limitations which make them less than ideal for this kind of research.

The language *Acore* is being designed and developed to to help meet this need and facilitate future research. The name 'Acore' is derived from the idea that it is an *a*ctor *core* language. It is an actor language in that it is based on the actor model of computation, a model of concurrent computation which is well suited to expressing diverse concurrent processes It is a core language in that it is meant to be a base for future development of concurrent languages. Design of concurrent languages is an area of active research; Acore serves as a base from which aspects of language design can be developed in many directions.

1.1 Contributions of this Thesis

The contributions of the research presented in this thesis can be briefly listed as follows:

1. A set of major goals for programming languages which are suitable for experimenting with concurrent systems has been collected. Among these are the ability to easily express simple cooperative and competitive relations between concurrent processes, the ability to control processing by subprocesses, and the ability to abstract useful cliches both procedurally and syntactically.
2. Since these goals are unsatisfied by other concurrent languages, I have designed the language *Acore* to serve as a foundation for exploring techniques for using concurrent processes and ways of expressing these techniques. In this design I show

    a. how the *replacement behavior model of state change* of the actor model of computation can be extended to a higher level language through implicit use of the insensitive behavior. This permits immutable environments, so subexpressions may be evaluated concurrently and concurrent algorithms may be expressed concisely without sacrificing the ability to express state changes.

    b. how the *sponsor model for controlling processing* can be incorporated into a con-

current programming language.

c. how the features of function applying and message passing programming languages can be combined into one uniform language, using a common mechanism for behavioral abstraction as well as for invocation.

d. how competitive concurrency may also be expressed in an expression oriented way (using the *race* expression).

3. I show how Acore behaviors can be compiled into behaviors of primitive actors using concurrent continuations. I have developed a language, called *Pract*, for describing these primitive behaviors.

4. I have developed a compiler which performs the transformation from Acore to Pract, and explain the strategies used by the compiler for researchers in other communities who are interested in developing similar languages.

5. I have developed the Apiary Emulator for running compiled Pract programs, and Traveler, the Apiary Observatory for analyzing and debugging concurrent actor programs.

1.2 Organization of this Thesis

This thesis is divided into four parts. Part One is an introduction to the ideas of actors and the Acore language. Part Two takes a top down view on the design of Acore, first describing the highest level goals before proceeding with the details of Acore itself. Part Three takes a bottom up view on the implementation of Acore, first explaining the strategies for implementing Acore behaviors in terms of much more primitive behaviors, then describing how the Acore compiler translates Acore into the primitive actor language Pract. Part Four compares Acore with widely known concurrent languages, pointing out the shortcomings which motivated the design of Acore.

There are also several important appendices. Appendix A summarizes the design and implementation of the Apiary emulator for running compiled Acore programs. Appendix B describes the design and implementation of Traveler, an observatory for observing and debugging Acore programs. These are included since their development has influenced the design of Acore, and they help give a more complete picture of some of the Acore programming environment.

Full explanations of Acore's semantics and grammar are not presented in the body of this thesis; I refer readers who are interested in more fully understanding the language to the *Acore Reference Manual* included in Appendix C. Similarly, readers who are interested in a better understanding of Pract are referred to the *Pract Reference Manual* included in Appendix D.

Chapter Two

Actors and Acore

The actor model of computation is based on the idea that concurrent computation can be modeled as a system of communicating objects called *actors* ([Hewitt 77], [Agha 86], [Clinger 81]). It is well suited for modeling distributed processes in several aspects.

- Actors respond to their messages using only their local state, which models the physical separation of objects in a distributed process.
- Actors communicate with the asynchronous, buffered protocol of the mail system abstraction, which models the delay and arrival order nondeterminism of communication in distributed processes.
- Actors may create new actors, allowing dynamic allocation and process creation to fit the task at hand.
- Actors have indefinite lifetimes and global extent, permitting arbitrary sharing between concurrent processes.
- Actors may change their behavior, so history sensitive objects may be modeled and shared between concurrent processes.

These properties also make it well suited as a model for the concurrency of a concurrent programming language. In this chapter I will introduce the central concepts of the actor model of computation through examples which introduce Acore.

2.1 An Actor

An *actor* is an object which may receive messages and send more messages in response, such as the simple bank account described in figure 2-1. A *script* defines the behavior of an actor; simple-bank-account is a script which defines the behavior of the actor my-account. An actor may have *acquaintances*, other actors which the actor knows as part of its local state, such as the bank account's balance. In Acore, a *message* consists of a *keyword* and zero or more parameters. The keyword selects one of the *handlers* of the script; for example, a message with the :balance keyword sent to an actor with the simple bank account behavior will invoke the :balance handler.

Each handler must specify a *replacement behavior*, the behavior the actor will have in processing the next message it receives. For some handlers, the next behavior will always be identical to the current behavior; these are called *unserialized* handlers. For example, the :balance handler is unserialized. An actor whose behavior consists entirely of unserialized handlers is an *unserialized actor*; since it never changes state, its messages needn't be processed in any serial order. For other handlers, specifying the replacement behavior may just call for specifying new acquaintances. The *ready* command specifies

```
(DefName simple-bank-account                    ; definition of a bank account behavior
  (script (balance)
   ((:balance () :unserialized)
    balance)
   ((:deposit (amount))
    (ready (balance (:+ balance amount)))
    self)
   ((:withdrawal (amount))
    (let ((new-balance (:- balance amount)))
     (if (:< new-balance 0)
      (then (ready)
            (complaint :overdraft))
      (else (ready (balance new-balance))
            self))))))
                                                ; definition of a bank account
(DefName my-account (:create simple-bank-account 0))
```

Figure 2-1: A simple bank account written in Acore.

that, with the specified changes to zero or more acquaintances, the actor is ready to receive the next message. For example, the *ready* command in the :deposit handler specifies that balance bound to the new balance for the next message, and with that change the actor is ready to process its next message.

The other commands within the handler follow a syntax similar to that of Lisp: *let* binds identifiers to intermediate values, *if* makes decisions between alternate paths, and the final expression determines the value to be returned. However, commands and expressions in Acore are performed *concurrently* by default, unless there is an ordering constraint (e.g. the identifiers of a *let* must be bound before the body is performed). It is not the purpose of this thesis to give a tutorial in Acore programming, so I won't spend much time explaining Acore syntax and semantics. If readers are confused by an example or wish to learn more about Acore constructs, I refer them to the *Acore Reference Manual* reproduced in Appendix C.

Now that we have an example of an actor in mind, let's look at the capabilities of actors more systematically.

2.2 Capabilities of Actors

As formalized by Gul Agha in [Agha 86], an actor consists of a *mailing address* which uniquely identifies the actor, and a *behavior* which specifies what the actor will do upon receiving its next message. Upon accepting a message an actor may concurrently:

- make simple decisions,

- create new actors,
- send new communications, and
- specify a replacement behavior

Simple Decisions: Simple decisions are decisions which can be made using only the information in the message and in the local state of the actor, such as whether one reference to another actor (a mailing address) is identical to another reference. In the bank account example, whether to perform the withdrawal or to complain is based on whether the reference returned from the expression testing for a negative balance is identical to a reference to (the actor representing) false.

Creating Actors: Creating an actor requires specifying its initial behavior; often the actor's new behavior will be parameterized with its initial acquaintances. For example, my-account is created with the simple-bank-account behavior an an initial balance of 0.

Sending Communications: Sending a message requires specifying the contents of the message and the mail address of the recipient, which is sometimes known as the *target*. The simple bank account sends requests to compute intermediate values; for example, (:< new-balance 0) is a request to new-balance with keyword :< and parameter 0. It also sends a reply with the final value, e.g. balance or self, to the *customer* of the request.

Specifying a Replacement Behavior: The *replacement behavior* specifies how the actor will respond to the next message received. Note that the replacement behavior must always be specified, or the actor wouldn't be able to process its next message. While the actor is waiting for its next behavior to be specified, it takes on the *insensitive behavior*, so any messages which happen arrive cannot be processed and must be buffered or queued.

Specifying a replacement behavior is how actors model change of state, such as the change in balance of the bank account. An actor may also more completely change its behavior, as in the simple locker presented in figure 2-2. This simple locker alternates between the empty locker and full locker behaviors. (The key script is just used to create a unique value — no messages are sent to key actors.) Being able to change state is the crucial feature of the actor model which provides actors with the capability to model extensible systems that can be adapted for changes without shutting down and rebuilding.

Every operation an actor makes uses only the local information the actor has at hand: information stored in its local state (including constants of its behavior) and information which is part of the incoming message. These operations may be sequenced due to data dependencies — for example, an actor may need to make a decision and create a new actor before it can send a message to the new actor — but otherwise the operations are concurrent. Identifiers in Acore are *referentially transparent* (they have a single value since bindings are immutable), and may easily be shared between concurrent operations; we'll come back to this point later. Specifying the replacement behavior is the *only* method of changing

```
(DefName Empty-Locker
  (script ()
   ((:store (valuables))
    (let ((my-key (:create key)))
     (:replace full-locker self my-key valuables)
     my-key))
   ((:retrieve (key))
    (ready)
    (complaint :wrong-locker))))

(DefName Key (script ()))

(DefName Full-Locker
  (script (my-key valuables)
   ((:store (new-valuables customer))
    (ready)
    (complaint :full-locker))
   ((:retrieve (customer-key))
    (if (eq? customer-key my-key)
     (then (:replace empty-locker self)
           valuables)
     (else (ready)
           (complaint :wrong-key))))))

(DefName my-locker (:create empty-locker))
```

Figure 2-2: A simple locker written in Acore.

state in an actor; there are no assignment commands.

Since actors may concurrently send many messages to distantly distributed actors, the communication model is important to keep in mind.

2.3 Communication

Communication between actors is performed using the *mail system abstraction*. Like mail systems in the real world, messages are sent asynchronously, and are buffered for the recipient. Messages may be delayed in transit, leading to an arrival order nondeterminism between concurrently sent messages, but since the mail system provides a *guarantee of delivery*, the delay may not be infinite.

Asynchronous communication means that an actor sending a message just gives the message to the mail system and goes on to other things. In particular, unlike synchronous communications, the actor does not wait for an acknowledgment that the recipient has accepted the message. Asynchronous communication allows actors to make use of the time during which messages are in transit. When

synchronous communication is required, an acknowledgment may be returned as a separate asynchronous message. Thus, we think of sending a request message to the bank account which later sends a reply message; the two communications are separate. Much of the time, however, some actor is waiting for the reply, since the value returned is important for later computation.

Buffered communication means that messages arriving at their recipient are queued until the actor is ready to accept them. This buffering is necessary due to the changing behavior of the recipient (thus the recipient may not be ready to accept the next message) and the asynchronous nature of the communications (the sender doesn't wait for the recipient to be ready). Since the actions of an actor may be performed concurrently, there may also be some pipelining between the processing of consecutive messages. When the actor is busy processing previous message, it cannot process another message until it computes its replacement behavior. As soon as the replacement behavior is known, the next message in the mail queue may be processed, even if messages are still being sent concurrently by the previous behavior.

In any shared communications medium such as a network, messages may be unpredictably delayed in transit due to traffic congestion. Thus messages which are sent concurrently may arrive in a different order than they were sent — there is *arrival order nondeterminism*. The mail system includes *arbiters* to decide an order for messages arriving at nearly the same time at any point in the communications network; they are the source of nondeterminism. Despite this nondeterminism and the possibility for arbitrary delays, the *guarantee of delivery* of the mail system can be used to prove termination for some actor programs.

2.4 Implications

Since concurrency in actor programs is engendered simply by sending multiple messages, concurrency comes naturally and inexpensively to actor languages, and a fine grain of concurrency results. Parallel computer architectures can take advantage of this fine grain of concurrency in balancing the processing load between processors by migrating actors. There may be no relation between the structure of the processor network and the structure of the program, and even less between the structure of the network and highly active, communicating parts of the program, so the actors of the program may be distributed among the processors in ways in which communicating actors may be separated on distant processors. The asynchronous, buffered nature of communications between actors simplifies the protocol necessary between the processors involved.

Thus, the actor model is suitable as a paradigm for thinking about concurrent, distributed computations. Since we are interested in experimenting with concurrent processes, we are interested in a language for expressing those processes, simplifying the task of creating the experiments. However, there are many issues in designing such a language; we turn next to the issues involved the design of Acore.

Part II
Design of Acore

Chapter Three

Design Goals

Many different design goals can effect the design of a new language, but good languages are designed with a few key ideas in mind, and the other goals follow from these ideas. The highest level goals in the design of Acore are generality, unification of object oriented and function oriented programming paradigms, forming a base for future development, and mathematical foundation. This chapter explains the motivation for these key ideas behind the design of Acore; the lower level goals which follow from them will be described in the following chapter on the design of Acore.

3.1 Generality for Expressing Concurrency

To explore the possibilities created by concurrent computation, we need a language which is capable of easily expressing diverse kinds of concurrent processes, including processes with cooperating, competitive, and resource limited sub- or co- processes.[1] Our language should be general enough to express these organizations within the paradigms encouraged by the language.

Cooperative processes work together toward a common goal, perhaps through simple division of labor as in divide and conquer algorithms. For example, a concurrent version of quicksort may use concurrent sub-processes to sort each half of the partitioned sequence; the partition function itself may use concurrent processes to partition each half of a sequence. This simple division of labor is a very common source of concurrency arising primarily due to lack of data dependencies between the concurrent processes. Therefore our language should be able to easily express concurrent algorithms as concurrency within functions and procedures.

Cooperative processes may also form more complicated organizations, such as in blackboard systems used in artificial intelligence. One example is a combinatorially implosive algorithm described in [Kornfeld 82], where concurrent processes working toward a goal share partial results through a common database (the "blackboard"). The combinatorial implosive aspect of the algorithm comes from the feature that the partial results entered in the database by one process may help reduce the search space of another process. Thus our language should be able to express objects with changing state, such as a database, shared between concurrent processes.

Competitive processes compete with each other to reach a goal. A simple example is a parallel

[1]The idea of a 'process' used here is not restricted to the sequential process which may be familiar from present computer operating systems, but is closer to the everyday meaning of the word. Thus processes can be composed of concurrent sub-processes. Similarly, a 'procedure' may be a set of steps to follow, but the steps may not have a total sequential order.

conditional with several branches, each guarded by a test. If each of the tests may take arbitrary amounts of time, the fastest way to select one of the branches may be to evaluate all of the tests concurrently, and take the branch corresponding to the first test to return positively. A similar situation occurs when there are several algorithms for reaching a goal, each of which is fast for some partition of the possible cases. The fastest way to compute the answer in general may then be to run all the algorithms concurrently, returning the first answer produced. Both of these examples involve situations where the timing of returned results makes a difference. Our language must be able to express this kind of competition.

The indeterminacy of the results produced by a program which expresses competitive concurrency may alarm some people. Our philosophy is that the world is naturally modeled by objects which change state and communicate asynchronously. Nondeterminism arises from the combination of mutability and the arrival order nondeterminism of asynchronous communication. Thus, a general purpose concurrent programming language should be able to express such models so programs will be able to interact with such a world, as well as simulate it. In situations where worrying about the complex interactions possible in a nondeterministic system is an unnecessary hindrance, a deterministic (e.g. functional) subset of the language may be used.

Processes may also be both cooperating and competing to different degrees. For example, exploratory processes searching a database may return information which will be processed in the order it is found and returned. The processes are cooperating in that they are all searching different parts of the database, but they are also competing in that the order in which answers are returned may make a difference.

Sometimes it is necessary to control the resources used by competitive processes. In the simplest case, a process which is no longer needed may need to be stifled. For example, when several processes are competing to find a single answer, once the answer is found the remaining processes are no longer needed. If letting them continue would consume large amounts of resources, it may be desirable to stifle them and conserve those resources. Our language should permit at least such gross control over processes.

Finer control over resources available to concurrent processes is also desirable. Consider a situation where many competing processes are gathering evidence for or against different alternatives for a decision. In some instances of this situation it may be desirable to ensure that both sides have equal access to resources (e.g processor time). In other instances there may be a history indicating which alternatives are the most promising, yield the most information, or converge most quickly, so it may be desirable to initially focus resources on these processes. Therefore, a general purpose language for concurrent procedures should have the ability to express this type of resource control over processes.

In summary, one design goal for Acore is to ensure that it is general enough to easily express at least cooperative, competitive, and resource controlled process organizations.

3.2 Unification of Object Oriented and Function Oriented Programming Paradigms

The object oriented and function oriented programming paradigms have attracted much attention and popularity in the past decade or so. The object oriented, message passing programming paradigm is useful for applications which are most easily expressed with a rich set of datatypes and generic operations on those datatypes. The functional, lambda calculus based programming paradigm is useful for applications which are most easily expressed with a rich set of functional abstractions, including higher order functional closures created at run time.

Both of these paradigms depend on the idea of a garbage collected heap allocation for small structures of indefinite lifetime: the objects in an object oriented language, and the closures (and data structures) in a function oriented language. Yet these two paradigms form a complementary pair: current object oriented programming languages don't allow higher order behavioral abstraction, and current languages based on the lambda calculus don't facilitate a data driven programming style.

In current object oriented languages such as Smalltalk [Goldberg 83] or Flavors [Weinreb and Moon 81], object behaviors must be part of a class hierarchy. Methods may only defined at top level; a higher order behavior method which returns new behavior is not possible. In Smalltalk there are no anonymous functions; a class and a method name must be specified for every function created. The behavior of higher ordered functions can be simulated by creating a class for each lambda, explicitly specifying the free variables of the lambda as instance variables of the class, and supplying the values of the free variables when the instance is created. Thus the problem is not so much one of expressive power as one of ease of expression -- people don't think of using higher order functions in Smalltalk because they are difficult to express.

In current function oriented languages such as the Scheme dialect of Lisp [Rees et al. 86], a data directed message passing style of programming may be implemented using closures as objects, but it does not result in a uniform system. Objects which are primitive to the system, such as functions, lists, and numbers, do not have a message passing interface, while user defined objects do. Therefore the benefits of the data directed message passing style do not extend to the common primitive objects in the system, and these have to be handled as special cases in programs which deal with objects in general, such as procedures for printing objects or sorting collections of objects.

Therefore, a second major design goal of Acore is to unify these two programming paradigms, producing a language which can express higher order behaviors and which has a message passing interface to all objects.

3.3 Forming a Base for Future Development

Concurrent language design with the above goals is a new area of research, and designs for high level languages with these goals can still develop in many directions. In any one design for a high level language, many different choices must made about the abstractions provided by the language (and its library). Some choices will be made in terms of what functions, objects, operations on those objects, etc., will be provided by the language system. Other choices will be made about how the system is organized, what protocols are used, etc. More important decisions will be made concerning what abstraction mechanisms are provided, especially abstraction building abstractions for describing new behaviors, describing combinations of behaviors (e.g. through inheritance or delegation), or for incorporating description oriented declarative programming.

All these choices represent open topics for further research, yet all the languages based on the above goals may have a common core. Therefore a goal in the design of Acore is not to design a full blown high level concurrent programming language. Rather than take a big step and make many arbitrary choices all at once, Acore will be designed as a core language to serve as a foundation for the development of higher level languages. As such, it will implement only the primitive abstraction mechanisms: a behavioral abstraction mechanism which abstracts the procedures objects follow, and a syntactic abstraction mechanism (a macro facility) for abstracting syntax. With these abstraction mechanisms, research into many of the choices mentioned above can be made without changing or re-implementing the core compiler. Also, since the developed languages will be based on a common core, they will potentially be compatible with one another and can share the implementation of useful lower level tools, e.g. for editing, debugging, metering, etc.

3.4 Mathematical Foundation

Basing an aspect of a language's design on a good mathematical model is one way of gaining confidence that that part of the design is consistent and well thought out. Looking at the language from the viewpoint of the model can help designers, implementors, and users — the designer gains insights into how important details of the design can affect its semantics, the implementor may gain insights into details not strictly specified by the designer, and the programmer using the language gains insights into the expressive powers of the language. Deviations from the model may compromise the effectiveness of the modeling, and should be minimized.

Therefore a goal in the design of a new language is to choose or develop mathematical formalisms which reflect the other goals of the design. In our case, models which can express behavioral abstraction and invocation, and models which can describe diverse kinds of concurrent processes are desired.

Toward this end, the aspects of the Acore design concerning functions/procedures and name scoping have been strongly influenced by the the lambda calculus, largely through the influence of Scheme [Rees et al. 86]. The Actor model of computation is the source of ideas for modeling concurrent

processes as message passing objects [Hewitt 77] [Agha 86] [Clinger 81]. I will assume the reader is familiar with Lisp-like languages such as Scheme and the idea of message passing objects. We will look at these issues and many more as we delve into the design of Acore in the next chapter.

Chapter Four

Issues in the Design of Acore

During the process of taking a set of high level goals and creating a design for a language which meets those goals, many issues arise about which design decisions must be made. This chapter describes many of the issues which arose in the design of Acore, and the reasons for the decisions which were made. Too many decisions were made to cover them all here, so I can only hope to cover the most important ones.

Since the discussion in this chapter is largely concerned with the semantic issues in the design of Acore, a prefatory remark concerning syntax is in order. The syntax of Acore is in a parenthesized style very similar to Lisp. This style of syntax was chosen for three basic reasons. First, Acore was envisioned as a very expression oriented language, similar in this respect to Lisp, and full parenthesization is desirable to help make clear the complex structure of deeply nested expressions. (Contrast this approach with one where many intermediate values are assigned to variables.) Second, the simple, uniform syntax makes the language easy to extend by using macros. Third, using parenthesized syntax allows implementors (such as myself) to take advantage of existing parsers (readers) and development tools (e.g. editors) for Lisp.

The first sections of this chapter deal with the basic forms for expression and abstraction used in all Acore programs. They are followed by sections about the expression of behaviors which do not fit into the basic applicative order form: behaviors which do not return values, behaviors which use future concurrency or delayed evaluation, behaviors using competitive concurrency. Competitive concurrency leads to issues of controlling processes and exception handling. Finally, the chapter closes with discussions of top level naming, and the macro facility for extending Acore syntactically.

4.1 Expressions

The basic idea for the design of Acore is to take the ideas of message passing communication, state change, and concurrency from the Actor model, and define an interpretation of the expression oriented syntax of the lambda calculus, Lisp, and Scheme traditions in terms of a set of message passing primitive actors. Under this interpretation, message passing divides into the categories of *request* and *response*. Applying a function corresponds to sending a request message to an actor representing the procedure; returning a value corresponds to sending a response message to the waiting customer. (Primitive actors will be discussed in more detail in Chapter 6.)

4.1.1 Syntactic Issues: Ask Expressions

The basic unit of expression in Acore is the *ask expression*, which has the following form:

(<*message-selector*> <*target*> <*message-parameters...*>)

For example:

```
(:deposit my-account 100)
```

The message selector evaluates to a keyword symbol which selects which of the target's message handlers will be invoked. Keywords are actors like all other objects in the system. We have chosen to put the message selector first because in our experience writing Acore code we have found that in most cases the selector is a short constant such as :deposit. The reason for this regularity is that in most procedures the operations are fairly constant — only the actors involved need to be computed. In message passing style code, the operation is usually represented by the message selector. Larger expressions may be involved in computing the target or any of the parameters, so if the target were placed first as in many object oriented languages, the message selector becomes obscured in the middle of the ask expression. For example, compare the readability of the code fragments from a version of quicksort in figure 4-1. In the infix version, the operation tends to become lost between the nested subexpressions.

Infix:

```
((quicksort :do
  (left-lessers :append right-lessers))
 :append
 (quicksort :do
  (left-greaters :append right-greaters)))
```

Prefix:

```
(:append
 (:do quicksort
  (:append left-lessers right-lessers))
 (:do quicksort
  (:append left-greaters right-greaters)))
```

Figure 4-1: Infix vs. Prefix message selectors.

One of our major goals in designing Acore is to unify function oriented and object oriented programming styles. Since invocation corresponds to sending a request message, functions (closures) are represented by actors as well. We have introduced the convention that functions are actors which have a handler for the :do message. An inconvenience arises when writing and reading code which uses many functions: most of the message selectors are :do, and become pretty meaningless, while the function names are names of operations. To alleviate this inconvenience, :do keywords are optional and may be

omitted. Thus, if the message selector is not a keyword constant, the ask expression is interpreted as a function call by inserting the message selector :do. For example, compare the readability of an extreme example from a function oriented version of factorial in figure 4-2 with and without :dos.

With the :do selector

```
(if (:do = n 0)
 (then 0)
 (else (:do * n (:do factorial (:do - n 1)))))
```

Without the :do selector

```
(if (= n 0)
 (then 0)
 (else (* n (factorial (- n 1)))))
```

Figure 4-2: ':do' can reduce readability.

This is a compromise — at first glance it appears this approach sacrifices the ability to use an arbitrary expression for the keyword. We noted that most of the time the keyword is a constant, but there are still times when you must be able to specify an expression for the keyword. Upon closer examination, however, a solution to this dilemma appears. Expressions which appear in the message keyword position of the ask expression are treated as the target of the ask. Since message keywords are objects, we may give them a behavior. In particular, we can give them the behavior of a function which accepts :do requests and sends the request which would have been sent had the keyword been a constant. For example, consider the following code fragment:

```
(let ((keyword (if positive? (then ':+) (else ':-))))
 (keyword value delta))
```

By our rule for inserting :do this is equivalent to

```
(let ((keyword (if positive? (then ':+) (else ':-))))
 (:do keyword value delta))
```

Suppose positive? is true, so the keyword is the symbol :+. The :+ symbol is then sent the message:

```
(:do :+ value delta)
```

So all :+ has to do is send the :+ message to its first parameter (value) with the rest of the incoming parameters (delta):

```
(:+ value delta)
```

This compromise sacrifices a little efficiency in this case, since an extra message must be sent, but we lose no expressive power and it is worth the extra readability in function oriented procedures.

So far we have looked at the issues behind the syntax and semantics of individual ask expressions. Next we shall examine how to build more complex expressions by composing ask expressions in several

ways. The value of an expression can be used as parameters to another expression or command, for making decisions about what other expressions to evaluate, or as part of a race for computing values.

4.1.2 Composing Expressions for Simple Division of Labor

When ask expressions are composed together by nesting and using *let*, several issues arise. One of our goals for Acore is that the concurrency inherent in algorithms be easy to express; in particular, the concurrency expressing the simple division of labor between independent expressions should be basic. Therefore, *expressions and commands in Acore are by default concurrent.* The nested expressions forming the parameters of an ask expression do not depend on each other, and are performed concurrently. Once all the parameter expressions have been evaluated, the ask expression itself can be performed (initiating a request to the target). Similarly, the expressions giving values for the identifiers in the arms of a *let* expression are independent of each other, and are evaluated concurrently. Once all the arm expressions have been evaluated, the commands and expressions forming its body are performed concurrently.

For example, let's take a look at our code fragment from quicksort, this time expressed with and without a *let* in figure 4-3.

Concurrent Subexpressions

```
(:append
 (quicksort
  (:append left-lessers right-lessers))
 (quicksort
  (:append left-greaters right-greaters)))
```

Concurrent Let Arms

```
(let ((sorted-lessers
       (quicksort (:append left-lessers right-lessers)))
      (sorted-greaters
       (quicksort (:append left-greaters right-greaters))))
 (:append sorted-lessers sorted-greaters))
```

Combination

```
(let ((lessers (:append left-lessers right-lessers))
      (greaters (:append left-greaters right-greaters)))
 (:append (quicksort lessers)
          (quicksort greaters)))
```

Figure 4-3: Equivalence of subexpressions and let arms.

The first two of these expressions are necessarily equivalent. In both cases, the (recursive) calls to quicksort are independent of each other, and the two quicksort expressions, including their nested sub-expression, may be evaluated concurrently. When both values of from quicksort have been returned, then the final append can be performed.

The third of these expressions is not equivalent to the other two. In the first two cases, each call to quicksort may proceed as soon as its parameter has been evaluated. However, in the third case, the calls to quicksort are held up until both of the preliminary appends have completed.

From this example it is evident that Acore performs applicative order evaluation of expressions by default. Eager evaluation[2] using *futures* would result in even greater concurrency. However, future concurrency is harder to understand and control, especially in the presence of actors which may change state. Also, it is occasionally desirable to sequence expressions. With applicative order evaluation, two sequential expressions can be expressed simply by putting the first in the arm of a *let* and the second in the body; this would be harder to express with eager evaluation. Futures also involve some additional overhead to represent the future, queue any premature messages, and forward all messages to the actual value. Therefore eager evaluation is less suitable as a default than simple applicative order evaluation.

To keep the semantics of many concurrent expressions under control, all identifiers in Acore are *referentially transparent*. No identifier ever changes value to another actor; instead, state change is modeled by the change of behavior. This is an important point which I will return in section 4.3.

4.1.3 Composing Expressions for Making Decisions

The primitive decision making capability of actors allows them to compare mail addresses to find whether or not they are identical. For making arbitrary decisions, this capability is most useful when coupled with the globally distributed knowledge about what mail address represents a boolean value, e.g. false. In other words, if all actors agree on a value for false, decision making may proceed on the basis of queries between actors which expect a boolean value (e.g. false or not false) to be returned. Once this convention is established, ask expressions can be used for making decisions using *if*. For example, consider again our factorial example:

```
(if (= n 0)
  (then 0)
  (else (* n (factorial (- n 1)))))
```

A request to the = function is made, and once the the reply is received, one of two paths may be chosen based on whether or not the value returned was false.

This method of making decisions was chosen because it is the simplest to understand. No complex concurrency issues arise because it is sequential: the test must be completed before either of the

[2]Eager evaluation is where an actor, a *future*, which represents the value is returned immediately so the value can be distributed concurrently with its evaluation. See section 4.6.

consequences is begun. Other evaluation strategies are possible; for example, a strategy which starts evaluating the consequences in parallel with the test overlaps the evaluation of the consequence with the test and may be faster. However, such an evaluation strategy introduces the problem of stopping evaluation of the consequence not chosen. Furthermore, in situations where actors change state as a consequence of the evaluation, the meaning of the choice is not clear. There may be situations where an overlapping strategy for decision making is desirable, and I encourage people to try out such ideas using Acore, but these complex consequences make such a strategy unsuitable as the primitive decision making strategy for Acore.

4.1.4 Composing Expressions Competitively

One of our design goals for Acore is that it be able to cleanly express competitive concurrency. However, competitive concurrency is a new capability offered by concurrent languages, and as such it is not well understood. In particular, we don't yet know what paradigms for competitive concurrency are most useful. I expect that Acore will be a useful tool for experimenting with competitive concurrency and learning what paradigms are useful; perhaps at a later date some of the most common paradigms will be incorporated into Acore itself. Given these considerations, an expression of competitive concurrency which captures the essence of all paradigms is desirable. People can then experiment with specialized paradigms by manipulating this form, perhaps using macros and auxiliary actors.

Competitive concurrency is expressed in Acore through the *race* expression, which has the following form:

(race expr1 expr2...)

The value of a *race* expression is a queue of values in the order that they are received. For example, an expression which returns the first answer received from two contestants is the following:

```
(:first (race (:ask contestant1 question)
              (:ask contestant2 question)))
```

I will discuss competitive concurrency in more detail later in this chapter; for now it suffices to introduce it as an additional method for composing expressions. Next we take up the issues involved in encapsulating expressions into a behavior.

4.2 Behavioral Abstraction

A major goal in the design of Acore is to unify the function oriented and object oriented programming paradigms. As a result, the abstraction mechanism of Acore takes features from the abstraction mechanisms of both paradigms. Like the closure oriented paradigm, behaviors can be created by an expression at any point, and closed over any environment. Like the object oriented paradigm, abstraction is divided into two parts, defining a general behavior and creating an instance of it. In addition, as in the actor model, behaviors may be used for specifying the replacement behavior of an actor.

4.2.1 Syntactical Issues: Script Expressions

The *script* of an actor defines its general behavior: what messages it is prepared to handle, and what acquaintances it has. A *script* expression has a form as specified in figure 4-4. For example, the script of a salesperson's record might be defined as in figure 4-5. (*Unserialized* means the handler doesn't change the state of the actor.)

Form of script expressions:

(*script* (*acquaintance names...*)
  message handlers...)

Form of normal message handlers:

((*message-selector* (*message parameters...*) *handler options...*)
  expression-body)

Figure 4-4: *Script* syntax.

```
(defname salesperson-record-behavior
 (let ((department sales-department))
  (script (name salary commission)
   ((:name () :unserialized)
    name)
   ((:pay (sales) :unserialized)
    (:+ salary (:* sales commission)))
   ((:supervisor () :unserialized)
    (:manager department)))))

(defname my-record
 (:create salesperson-record-behavior
        "Carl Manning" 150 .07))
```

Figure 4-5: A simple script for a salesperson's record.

Several issues have arisen concerning the syntax of the script expression. First, why should the behavior be separated into handlers? Why can't a single lambda expression do? Second, why are the message parameter lists limited to simple lists? Why don't we allow more structured matching, as found in logic languages such as Prolog?

The script expression isn't directly analogous to a lambda expression, since a lambda expression

creates a complete closure which may be applied to parameters to produce a result, whereas the script expression only produces a behavior which may be used to create an actor which then can be sent a message to produce a result. The script expression separates the specification of the behavior from the creation of the actor, so that actors with the behavior can be instantiated in places other than where the behavior is expressed.

The script expression is separated into handlers to simplify the very common case where the handlers are indeed separate. In this common case, the visual separation of the handlers increases readability, and the parameter lists conveniently destructure the messages automatically instead of forcing the programmer to do so for each message. An otherwise handler, which processes messages not selected by any other handler, can be used if there is need to revert back to an all-in-one specification.

Handler selection isn't done by structured matching of messages primarily for efficiency reasons. Since handlers are selected on the basis of only the message selector keyword, handler selection can be very fast. Since parameter lists are flat, standard procedure calling techniques for binding values can be used.

Another reason for discouraging handler selection by structured matching appears if you consider implementing an actor system on a distributed architecture. Structured matching is often used on a deeply nested structure, even if the actual pattern is shallow, and programs which use it recurse through the structure. Therefore, either the entire structure must be transmitted as the message between processors of the distributed architecture, or there must be a message passing interface to the structure. The first is unacceptable due to the overly large communication overhead, and the second sacrifices the primitiveness of structured messages — structured messages can't be primitive if message passing is required to destructure them. However, a language which does destructure messages this way could be developed using Acore, and I encourage readers to try it.

4.2.2 Creation, Replacement, and Initialization

Two issues arise concerning the use of behaviors: How should they be used, i.e. what should be the value of a script expression and how should it be used, and how should initialization be performed, i.e. how should the initial acquaintances of an actor being created with or taking on this behavior be specified?

The first issue concerning use of script expressions is the question of just what a script expression should return. One possibility (which we've tried) is for the expression to return the raw representation of the script. Then actors can directly create more actors with this script, and specify replacement behaviors using this script. This mimics the raw handling of behaviors in actor theory, and allows efficient creation and replacement, but raises two problems of robustness in the implementation. First, the implementation may be sensitive to the relationship between the script and the actor; exposing the script either limits the kinds of efficiency tricks implementations can use or can impose implementation

dependent limitations on how scripts may be used in programs. Second, checking for the correct number of acquaintances is necessary for the robustness of the system, but to find the number of acquaintances the script expects requires message passing. Thus, if message passing is required anyway, the script might as well be encapsulated. Therefore we have turned to a second possibility, encapsulating the script in a guardian actor, as the default case (though we can return to the first method for optimizing critical code).

A script expression returns an actor which serves as guardian for the script. This guardian accepts :create messages to create new actors with the script, and :replace messages which specify a new replacement behavior with the script. (I will discuss change of state in more detail later.) Guardians encapsulate the script, hiding the implementation details of how scripts are implemented and how they are related to the actors with the behavior specified by the script. The message handlers for the guardian's :create and :become messages require the correct number of acquaintances, so the need for checking this is satisfied.

Another issue concerns initializing an actor with a behavior. There may be need for checking that the acquaintances are of the right type, or perhaps some acquaintances should be initialized automatically. One possibility is to find a way to incorporate this initialization information into the script expression. However, initialization can be an arbitrarily complex procedure, so rather than complicate the relatively simple concept of the script expression, initialization may be specified by encapsulating the script expression in a custom guardian actor which performs the necessary checking and initialization. For example, consider the safe account behavior of figure 4-6. It checks that the minimum balance is adhered to when creating a new account, and initializes it with an interest rate. It also checks to see that the minimum balance is adhered to when specifying this as the replacement behavior for an account, and that the interest rate isn't below standard.

4.2.3 Closure over Identifiers

The values of the free identifiers in a script expression are captured in a lexically scoped manner familiar to languages such as Scheme. For example, in figure 4-5, the department is captured as a free identifier, and will always refer to the sales-department in all actors with this behavior. Lexical scoping permits intuitive results from the nesting of script expressions. One consequence is that lambda expressions may implemented as in figure 4-7. The actor returned by a lambda expression is a function, an actor which accepts a :do message. A macro can easily be written to make this translation.

4.3 Referential Transparency of Identifiers

Acore expressions can be deeply nested, highly concurrent, and abstracted into lexically closed behaviors. Since they represent concurrent processes, they may be evaluated on distributed processors. Several problems arise if an identifier were to change value.

```
(defname safe-account-behavior
 (let ((minimum-balance 5.00)
       (initial-interest 0.05)
       (account-script
        (script (balance interest-rate)
         ((:balance () :unserialized)
          balance)
         ((:interest-rate () :unserialized)
          interest-rate)
         ((:deposit (amount))
          (ready (balance (:+ balance amount)))
          self)
         ((:withdrawal (amount))
          (let ((new-balance (:- balance amount)))
           (if (:< new-balance 0)
            (then (ready)
             (complaint :overdraft))
            (else (ready (balance new-balance))
             self)))))))
   (:create                                   ; create the guardian for the account script
    (script ()                                ; define the guardian's behavior
     ((:create (balance) :unserialized)
      (if (:< balance minimum-balance)
       (then (complaint :below-minimum-balance))
       (else (:create account-script
                      balance initial-interest))))
     ((:replace (actor balance interest) :unserialized)
      (if (or (:< balance minimum-balance)
              (:< interest initial-interest))
       (then (complaint :bad-initialization))
       (else (:replace account-script
                      actor balance interest))))))))

(defname my-account (:create safe-account-behavior 100.00))
```

Figure 4-6: Safe account behavior with checking and initialization.

First, when there are many concurrent expressions which share use of an identifier, if one expression is allowed to change the value of the identifier through assignment, the behavior of the other expressions may be unpredictable unless they synchronize reading the identifier with the assignment. However, providing synchronization at the level of identifiers greatly complicates the interpretation of concurrent expressions, and can obscure code.

Second, when distributed processes are sharing an identifier, it improves locality of reference if the binding of the identifier may also be distributed. If identifiers were able to change value, then the binding must be centrally located, e.g. in an environment.

The lambda expression:

```
(lambda (increment)
  (lambda (n)
    (+ increment n)))
```

is expressed in raw Acore as:

```
(:create (script ()
          ((:do (increment) :unserialized)
           (:create (script ()
                     ((:do (n) :unserialized)
                      (+ increment n)))))))
```

Figure 4-7: Expression of lambda using scripts.

Due to these problems, identifiers in Acore are *referentially transparent*, i.e. they do not change value. Therefore Acore is designed to have only *referentially transparent* identifiers. Identifiers may have only one value; their bindings are immutable. This imposes some constraints on the design of Acore. There is no assignment command in Acore; Acore follows the actor model in providing state change only through change of behavior (see the following section). Iteration must be performed through recursion.

The benefits of referentially transparent identifiers are threefold. First, concurrent expressions needn't worry about the values of identifiers changing out from under them. Second, bindings of identifiers can be freely distributed to promote parallel processing, copied into the state of the behaviors which use them. And third, related to the second, closures over identifiers may be implemented in an entirely local manner. Since bindings never change, each closure may keep its own copy of the binding.

The referential transparency of identifiers can also be helpful in constructing proofs about the behavior of actors. Together with the encapsulation provided by actors, i.e. the fact that they only respond to messages and their state cannot be accessed or modified by arbitrary agents from the outside in any other way, referential transparency of identifiers means invariants in the behavior of actors may be checked more easily.

Thus, referential transparency of identifiers keeps Acore consistent with the actor model conception of computation through distributed actors processing messages while referring only to their local states. But if identifiers never change value, how are state changes expressed in Acore? We take up this topic in the next section.

4.4 State Change

Behaviors of objects in Acore are patterned after behaviors in the actor model of computation. For example, in the actor model, the actions of the handler may be performed concurrently; the same is true in Acore. As we've described it so far, Acore has extended the expressive power of handler bodies by introducing ask expressions, permitting complex computations to be expressed in the body of handlers. Thus, it should be no surprise that Acore models change of state as specification of replacement behavior, only extended by allowing the replacement behavior to be computed using ask expressions.

The advantage of this approach to modeling change of state is that we can preserve referential transparency of identifiers. When an actor specifies its replacement behavior, it specifies how the *next* message it receives will be handled. It specifies the script and the bindings of acquaintance names to be used for processing the *next* message; it does **not** affect the bindings of names used for processing the current message. Once the replacement behavior is specified, the next message may be processed in the context of the new bindings.

This approach to specifying change of state syntactically sugars the common practice of assuming the *insensitive behavior*. Until the replacement behavior is computed, the actor assumes the insensitive behavior, queueing any messages it may receive. Once the replacement behavior is known, then the next message may be processed.

```
(DefName simple-bank-account                    ; definition of a bank account behavior
  (script (balance)
   ((:balance () :unserialized)
    balance)
   ((:deposit (amount))
    (ready (balance (:+ balance amount)))
    self)
   ((:withdrawal (amount))
    (let ((new-balance (:- balance amount)))
     (if (:< new-balance 0)
      (then (ready)
            (complaint :overdraft))
      (else (ready (balance new-balance))
            self))))))
                                                ; definition of a bank account
(DefName my-account (:create simple-bank-account 0))
```

Figure 4-8: A simple bank account script.

For example, consider again our simple bank account, reproduced in figure 4-8. The :balance handler is unserialized, so the account immediately assumes the same behavior as the replacement behavior. The :deposit and :withdrawal handlers are serialized; the *ready* commands within them

specify the binding of balance to be used in processing the next message received by the account. For convenience (and implementation efficiency), *ready* commands are used when the replacement behavior has the same script; only the values of zero or more acquaintances are changed.

```
(defname stack-node-behavior
 (script (value next)
  ((:element () :unserialized)
   value)
  ((:next () :unserialized)
   next)))

(defname stack-behavior
 (script (Top)
  ((:push (new-value))
   (ready (Top (:create stack-node-behavior new-value Top)))
   new-value)
  ((:pop ())
   (if (:null? top)
     (then (ready)
           (error "Empty Stack!"))
     (else (ready (Top (:next Top)))
           (:element Top))))))
```

Figure 4-9: The stack behavior in Acore.

Consider now how a stack behavior may be written in Acore: see figure 4-9. Focus on the final two lines of the stack behavior. The *ready* command specifies the value of the acquaintance Top for the next message. The last line makes an :element request to the top node to reply as the value for this request. Although these two lines are concurrent, there is no confusion concerning the value of Top in this context since identifiers are referentially transparent. Thus, referentially transparent identifiers afforded by this model of state change simplifies the dealing with changes of state in concurrent processes.

In either of these examples, as soon as the ready command has been performed, the actor (the bank account or the stack) may concurrently start processing its next message, even while it is still finishing up the previous message. Not only is there concurrency between expressions for handling a single message, but there may also be concurrency between handling successive messages. Thus, the replacement behavior model of state change allows *pipelining* the processing of messages to an actor, easing bottlenecks.

Finally, let's take another look at the locker behavior in figure 4-10, which we've seen previously. *Ready* commands are used when an actor keeps its current script. When it changes behavior completely and takes on a new script, it requests the new script to :replace its old behavior with a new behavior,

```
(DefName Empty-Locker
  (script ()
   ((:store (valuables))
    (let ((my-key (:create key)))
     (:replace full-locker self my-key valuables)
     my-key))
   ((:retrieve (key))
    (ready)
    (complaint :wrong-locker))))

(DefName Key (script ()))

(DefName Full-Locker
  (script (my-key valuables)
   ((:store (new-valuables customer))
    (ready)
    (complaint :full-locker))
   ((:retrieve (customer-key))
    (if (eq? customer-key my-key)
     (then (:replace empty-locker self)
           valuables)
     (else (ready)
           (complaint :wrong-key))))))

(DefName my-locker (:create empty-locker))
```

Figure 4-10: A simple locker written in Acore.

initialized with the necessary acquaintances. For example, when my locker is an empty locker and receives a :store request, it requests the full locker script guardian to replace its behavior with the full locker script parameterized with the valuables and key as acquaintances.

4.5 Commands

Generality is one of the major goals in the design of Acore. The expression oriented context of the handlers of scripts we've presented so far has been expressive and concise. However, not all behaviors can be expressed in terms of expressions, where "expressions" are forms which return a value. In particular, expressions cannot express handlers which do not return a value. Such handlers occur whenever an actor needs to queue an incoming message without processing it immediately; one example of such a handler is in the insensitive behavior of a *future* (figure 4-11), an actor which buffers all messages it receives and forwards them to another actor once the actor is known (see next section). The one alternative in expression context is to make an ask to an actor which does not return a value, but this just begs the question: now how can we define the behavior of *that* actor?

Therefore, although expression oriented handlers will conveniently express most behaviors people are likely to need, command context is also available in Acore for the rare cases where a value should not be returned and therefore cannot be expressed by an expression. Command context is as well developed as expression context: there are handlers with command bodies, commands for sending messages, and commands for specifying replacement behavior. *If* and *let* take on command bodies when found in command context. Ask expressions can be used computing the values of the parameters to commands, so the expressiveness of command bodies is very similar to expression bodies — the only difference is that command bodies don't have a value. (See the *Acore Reference Manual* in Appendix C for details on the commands available.)

For an example of the use of command context, refer to figure 4-11. Recall that the *insensitive behavior* queues messages until the replacement behavior for the actor is known; it does not return values for the requests because the messages haven't been processed by the actor's sensitive behavior. Thus a future behavior is very much like an insensitive behavior. In the figure, the bodies of the *is-request* handlers are *command bodies*; forms in these bodies are interpreted as commands and no value is returned. Thus, although ask expressions are used in the handler bodies, the values they return are ignored, and no value is returned for the requests to the future; they will be returned after the message is forwarded to the value.

```
(defname future-behavior
 (script (queue)
  (is-request (:ready (value))
   (:replace forwarding-behavior self value)
   (:map queue (clambda (entry) (:forward entry value))))
  (is-request (otherwise-selector (&rest parameters) :unserialized)
   (:enqueue queue
    (:create queue-entry
       otherwise-selector parameters
       sponsor customer reply-keyword)))))
```

Figure 4-11: An explicit representation of a future behavior.

Commands may also be found in expression context. We've already seen one command, *ready*, in the handlers for the bank account behavior in figure 4-8 and the stack behavior in figure 4-9. Commands may be used within the *expression bodies* of handlers and *if* and *let* expressions as long as they aren't the last form; the last form gives the value of the expression body. The *ready* commands in figures 4-8 and 4-9 precede the expression denoting the value to be returned by the expression body in which they are found.

Commands and command context permit specification of a class behaviors which do not fit into a strictly expression oriented syntax since they return *no* values. We turn now to behaviors which return

values *before* they are computed, so that computation and distribution of the value may proceed concurrently.

4.6 Future Concurrency: Distributing a Value Before it is Computed

By default, ask expressions in Acore are evaluated in an applicative order: first the message selector, target, and message parameters are concurrently evaluated, then the message is sent to the target (analogous to applying a function), and finally a value is received after the target sends back a reply. This imposes a sequential ordering on the control flow which, as was noted earlier, keeps the conceptual model of control flow simple. However, there may be times when breaking this sequentiality can be an important extra source of concurrency.

The sequentiality of applicative order evaluation can be broken by introducing a level of indirection separating references to a value from the actual value. Once this is done, then the value may be distributed concurrently with its computation, or even before its computation, by distributing the indirect reference. The indirect references are represented by actors called *futures*; the concurrency arising from their use is sometimes called *future* concurrency.

A *future* is a promise for a value, and a future actor promises to forward any messages it receives to the value it represents once the value is known. Thus the behavior of future is similar to that of the insensitive behavior, queuing messages until it knows what to do with them. Once the value is known, the future forwards all messages it has received so far to the value, and becomes a forwarding actor which automatically forwards to the value any further messages it receives.

4.6.1 *Future* and *Delay*

Since futures are an important source of concurrency and may be used often, Acore provides two special expressions to facilitate using this type of concurrency: *future* expressions and *delay* expressions. Both of these expressions take a single parameter, the expression for computing the value, (e.g. (*future expression*) or (*delay expression*)) and immediately return a future for the value. The difference is that *future* begins concurrently evaluating its expression immediately, whereas *delay* begins concurrently evaluating its expression only upon receiving a message. *Delay* expressions still provide future concurrency, since as soon as its first message is received, it starts computing the value concurrently with any distribution which is being performed.

Since *delay* expressions don't compute a value until a message is received, they provide *delayed evaluation*. Delayed evaluation can be used for programming in a stream oriented style, and make it possible to represent infinite data structures (see [Abelson and Sussman 85] for a further exposition of this subject).

Using futures does not always increase the concurrency of a program, and in fact increases the

overhead to run the program, since the future must be created and any messages it receives must be queued and forwarded. Consider the two functions in figure 4-12. For clarity, we've introduced DefFunction and Cond* forms which aren't primitives of Acore but can easily be written as a macros. DefFunction defines a name to be a function actor, and Cond* is a multi-branch conditional which tests its branches sequentially.

```
(DefFunction CopyTree (node)
 (if (:null? node)
  (then node)
  (else (cons (future (copytree (:car node)))
              (future (copytree (:cdr node)))))))

(DefFunction RangeProduct (lo hi)
 (cond* ((:= lo hi)
         lo)
        ((:= lo (:+ hi 1))
         (:* lo hi))
        (else
         (let ((average (:floor (:/ (:+ lo hi) 2))))
          (:* (future (rangeproduct lo average))
              (future (rangeproduct (:+ 1 average) hi)))))))
```

Figure 4-12: Use of Futures: RangeProduct vs. CopyTree

Both functions are doubly recursive and "futurize" the recursive calls. Since cons does not need to send any messages to its parameters, CopyTree takes advantage of the future concurrency by overlapping copying of the two subtrees with returning a value, which may be further distributed or stored in a structure. On the other hand, RangeProduct immediately sends one of the futures a :* message, and in order to do the multiplication both values need to be computed. Neither of the values are distributed anywhere else, so futurizing is not beneficial in this case.

4.6.2 Other uses for Futures

Sometimes it is necessary to provide indirection for other reasons, for example, to define mutually recursive definitions. In this case it is necessary to create a future without specifying what its value will eventually be. This can be expressed in Acore by giving *future* no parameters, as in (*future*). Since a future behaves like an actor with the insensitive behavior, the replacement behavior of a future specified in this manner can be specified by sending a :replace message to a script. A future normally becomes a forwarding actor, so the common case is to send such a message to the forwarding script.

For example, a LetRec which allows recursive definitions in its bindings may be implemented as in figure 4-13. In implementation using futures, first the names are given values as undefined futures,

This recursive let form defines two circular lists:

```
(LetRec ((A (cons  1 B))
         (B (cons -1 A))
         (C (cons  0 C)))
 ...body...)
```

and may be implemented (e.g. via macro) using indirection:

```
(Let ((A (future))
      (B (future))
      (C (future)))
 (:replace forwarding-script A (cons  1 B))
 (:replace forwarding-script B (cons -1 A))
 (:replace forwarding-script C (cons  0 C))
 ...body...)
```

Figure 4-13: Implementation of a Recursive Let using futures.

then the futures become forwarding actors to the actual definitions. Note that the form

```
(:replace forwarding-script C (cons  0 C))
```

causes C, a future, to become a forwarding actor to the cons cell returned from (cons 0 C), producing an actor which behaves like a circular list of zeros.

4.6.3 Forwarding Issues

The rangeproduct example brings up an issue in designing a language with futures and indirect references such as forwarding actors. How should we think of futures? Does the future *become* the value, or is the forwarding actor visible to the programmer? This issue is tightly connected with implementation strategies for the language.

From a purely message passing point of view, the forwarding actor is equivalent to the value — sending a message to either one has exactly the same effects and produces the same results. However, there are two types of situations where the message passing view isn't enough: places where the reference itself is important.

One place is in the implementation of actors like small integers: since numbers are unserialized, for efficiency they are represented so they can be recognized by the reference itself. Arithmetic on numbers is then a matter of examining the reference and computing a new reference, so a number which receives the :* message can compute the reply by examining the references of itself and the message parameters. However, this representation scheme breaks down if the parameters may be futures or forwarding actors.

We can solve this problem by setting up an appropriate protocol in one of several ways. For binary operations (e.g. division), one possibility is to make use of the fact that in the script which takes care of the operation (e.g. the integer script) the recipient of the message can never be a forwarding actor — otherwise the script wouldn't have been invoked. Therefore, if the parameter turns out not to be of the expected type (e.g. an integer), then a reversed message may be sent to the parameter. If the reversed message fails, then there is a runtime type mismatch and an error is signalled.

For example, if we send the message (:/ (*future* 6) (*future* 2)) the target is <future 6>. The message may sit in the future's queue for a (short) while, but eventually it will be forwarded to 6. 6 receives the :/ message with parameter <future 2>. Upon inspecting the reference to the parameter, 6 finds it is not a number, and so it sends the reversed message to the parameter: (:/-reversed <future 2> 6). Eventually 2 receives the reversed message with parameter 6 and calculates 6/2, replying the value 3.

The other possible protocol is more general but requires more message passing and continuation creation. Arrange for every actor to respond with itself up receiving a special message, say the :self message, except for futures and forwarding actors. Then to calculate (:/ (*future* 6) (*future* 2)), when 6 receives the :/ message with the parameter <future 2> and finds the parameter is not a number, it sends (:self <future 2>). Since all futures and forwarding actors do not answer the self message but forward it on, the value of this expression cannot be a future or a forwarding actor, so if it is not the expected kind of reference (e.g. a number), then it must be an error.

The second type of situation where message passing equivalence breaks down is where references to actors are being compared, for example when testing an actor for membership in a set, or simply testing if a boolean value is false in a conditional. In this case, without futures or forwarding actors no message passing is necessary, whereas if there is a future or a forwarding actor, then something like the :self protocol sketched above must be followed. Actors could try testing for identity first before going through the :self protocol, but note that most comparisons, especially in the membership example, will be failures, so this will not alleviate the problem much. Therefore it is to the programmer's advantage to know when forwarded values are not being used; in this case the much more efficient direct comparison may be made.

People who are familiar with forwarding pointers in traditional sequential architectures may wonder why I am laboring over the indirection problem. In sequential processors (e.g. a Lisp Machine [Moon 85], [Moon 84]) the forwarding pointer is tagged as such and the hardware follows the indirection whenever it is encountered. However, there are several differences in a distributed architecture. First, there is more overhead in following the forwarding — the forwarding actor and the value to which it forwards may be spread out across the network, and message passing is required to follow the forwarding. Second, forwarding actors in active use may not be in fast local memory as often, since they may also be frequently referenced by another processor. Therefore, since the overhead of forwarding is greater, the advantage of not going through the protocol of following it is greater as well.

One technique which may help the problem is to tag references to futures and forwarding actors (not just the actors themselves) and invoke the :self protocol only on these references. However, since actors may specify arbitrary replacement behaviors, any actor could conceivably become a forwarding actor, especially in long-lived systems which need to be patched. Therefore either many other actors would also have to be so tagged, defeating the gain in efficiency, or limit the actors which can become fully equivalent to other actors to the system defined futures and forwarding actors. The choice in this case seems to be a philosophical issue — is it better to stick with a clean semantics clearly corresponding to the actor theory, or is it worth sacrificing the clean semantics for more efficiency?

In light of the facts that such hardware assistance doesn't yet exist (though it could be implemented in software) and that Acore is a core language for experimentation with language design, I have taken the conservative position of sticking to the clean semantics. Therefore, there is a predicate in Acore by which a reference to a forwarding actor may be distinguished from a reference to the value to which it forwards; this predicate reflects the primitive capability of actors to compare references. However, it is possible for some language which is built from Acore to choose to hide or at least discourage use of this predicate.

One final point: an argument may be made that a predicate which compares only references is necessary to ensure the robustness of some kinds of programs, especially system level code. The :self protocol depends upon the integrity of the actors, so code which uses it can be broken simply by an actor which doesn't return self, or even worse, a future which never receives its value — a possibility now that we've introduced behaviors which needn't return a value in the previous section. Therefore there will always be some level which needs this primitive predicate.

In pursuing the goal of being able to express many kinds of concurrent processes, I've introduced ways of describing behaviors which return no value in the previous section. In this section I've introduced *future* and *delay* as the means of expressing future concurrency and, in the case of *delay*, delayed evaluation. I've also discussed the issues which arise from the forwarding necessary for future concurrency, and concluded that it is better to stick with the theoretical semantics and keep the primitive predicate for comparing references. Next we proceed into the least charted waters of all, the subject of expressing competitive concurrency.

4.7 Competitive Concurrency

Generality in being able to express many kinds of concurrent processes is a major goal of Acore. One class of concurrent processes not addressed by many concurrent languages is those with competitive concurrency, where the order of results returned from subprocesses may be important.

Competitive concurrency can be used in many forms. A few examples of some obvious forms: a concurrent **or** (or **and**) expression which returns true (false) immediately after one of its subexpressions

returns true (false); a concurrent conditional which selects the branch of the first test to return true; a expression which returns the list of the values of subexpressions in the order that they were received; etc. The conditional forms are especially useful in conjunction with a timer for timing out when something starts taking too much time or no answer is received. There are other useful forms as well, so the challenge in the design of a core language such as Acore is to find a form which may be used to program the other forms.

If a good primitive form of competitive concurrency may be expressed in Acore, experimentation with other forms is possible. For example, a concurrent **or**, **and**, or concurrent conditional is not primitive because it throws away values. Acore instead provides the *race* expression which evaluates its subexpressions concurrently and returns a queue of the values returned. The queue is returned immediately, and values are added to the queue as they are returned, so they may be read from the queue as soon as they are received. The queue behaves as a list, all or any tail of which may be a future representing the list of values not yet returned; when the final value is returned, the list will be terminated with **nil** like all lists.

The syntax of the race expression is as follows:
(*race expr1 expr2...*)

Since *race* is a special form, it is not as general as possible; for example, it is not useful for concurrently searching a database and queuing all entries found to pass some filter. This type of situation may be better expressed by introducing an explicit queue and making requests to enqueue answers into it. However, race is very useful for writing macros, such as for the example in figure 4-14.

This implementation of a parallel cond using *race* and *delay* works as follows. Each of the tests is evaluated concurrently. If the test returns nil (false), then nil is returned to the race; otherwise, the delayed body corresponding to the branch is returned. First-one finds the first non-nil value in queue returned from race; this will be the delayed branch of the first test to complete and return true. (Note that this implementation depends on being the delayed branch not being equivalent to nil -- it is testing the reference to the branch without invoking the delay and causing it to be evaluated.) If no tests return true, then the delayed else-branch is returned. Finally, a :self message is sent to the delayed branch, causing it to be evaluated and returning its value.

Similarly, a parallel or which returns the first non-nil value may be implemented as in figure 4-15.

I am not proposing that these are the best ways to implement these forms. These examples are meant only to help illustrate the utility of the *race* expression defining and experimenting with in less trivial control structures using competitive concurrency.

Parallel cond and parallel or point out a possible optimization which can be made in many instances of competitive concurrency: once the first successful value is known, the others are no longer

A parallel COND such as:

```
(cond (test1  body1...)
      (test2  body2...)
      (test3  body3...)
      (else  else-body...))
```

may be implemented (via macro) as:

```
(:self (first-one
       (race (if test1
                 (then (delay (let () body1...)))
                 (else nil))
             (if test2
                 (then (delay (let () body2...)))
                 (else nil))
             (if test3
                 (then (delay (let () body3...)))
                 (else nil)))
       (delay (let () else-body...))))
```

where first-one *is:*

```
(DefFunction First-One (possibilities default)
 (if (:null possibilities)
  (then default)
  (else (let ((possibility (:car possibilities))
              (remaining-possibilities (:cdr possibilities)))
          (if possibility
            (then possibility)
            (else (first-one remaining-possibilities
                             default)))))))
```

Figure 4-14: Possible implementation of parallel cond using *race* and *delay*.

needed. Therefore it is possible to conserve resources if the the unneeded computations may be stopped. This brings up the an important aspect of competitive concurrency: the control of resources distributed between competing processes, especially processor time. This is the subject of the following section.

4.8 Sponsorship

Competitive concurrency can often be tuned though control over the progress of the competing processes. In this section I discuss the nature of the control needed, and introduce *sponsors* as a mechanism for achieving this control in Acore. Finally, an example illustrates how the method is expressed in Acore.

A parallel OR such as:

(or *test1 test2 test3*)

may be implemented (via macro) as:

(first-one (*race test1 test2 test3*)
nil)

Figure 4-15: Possible implementation of a parallel or using *race*.

Competitive concurrency is primarily concerned with improving response time. It is used in situations where trying several approaches concurrently may produce answers or partial results faster than running them sequentially. Given such goals, it is important to be able to control the resource use of the competitive processes, so that useless or unpromising, possibly non-terminating subprocesses don't squander large processing resources, slowing down other processes and defeating the goal of improving response time. The challenge in the design of Acore is to find a general method to control concurrent processes which will serve as the foundation for experimentation with process management in the applications written in Acore.

The approach taken in traditional operating systems of giving priorities to processes is not applicable to actor computations because there are no structures representing processes. The closest thing to a process is a message, but messages multiply in vast numbers and are short lived, quickly processed and gone.

Another approach is to associate control with the actors involved. However, this approach divides the problem along the wrong dimension — many actors may be shared between concurrent processes. Also, a large number of actors may be involved in any one (sub)process, so it is difficult to organize control over them all.

The idea of a concurrent process in actor systems is not so much concerned with the actors involved with the process as it is with the flow of control. Most actor computations can be characterized as *transactions*:[3] a request is made to an actor, which may spawn many subtransactions, may cause some effects, and may finally return a value. A *transaction* may be loosely defined as a request and any processing it causes to produce the response. This definition is loose because transactions may interact, and some processing may be shared between several transactions (shared subtasks), or it may necessary

[3]For lack of a better term, we use the word "transaction" to loosely mean a request and any processing it causes to produce the response. This is roughly related, but not identical, to definitions of transactions used in other areas of computer science, such as for transaction processing in shared databases.

for many transactions but not clearly part of any one transaction (overhead).

As a result of this view, the approach taken in the design of Acore is to permit programs to manage processes in terms of transactions. Transactions consume resources, especially processing resources; therefore one way to manage processes is to control the amount of processing they use. In Acore, the managers are actors called *sponsors*, and the resource is processing *ticks*. Ticks are units of processing time corresponding to transactions; one tick is charged for every transaction. Transactions have subtransactions which are charged for as well, so this amounts to charging one tick for every request.

Therefore, every transaction must have a sponsor, and this sponsor is carried in the request. Conceptually, every time the system delivers and processes a request, it charges a tick to the sponsor of the request. If the sponsor runs out of ticks, it may, depending on the management structure of which it is a part, request more ticks from a parent sponsor. If no more ticks are available, then the transaction must be *stifled*. Stifling invokes the exception handling mechanism of the transaction — exception handling is very important for cleaning up stifled transactions, and will be discussed in the next section. At the very least, the exception handling for a transaction should release any insensitive actors which are awaiting the result of the transaction in order to determine their next behavior; usually the easiest thing to do is revert back to the previous behavior. The system charges only for the requests so that an exceptional response may be returned even if sponsorship has been depleted.

Sponsorship becomes an important consideration primarily when considering strategies for controlling competitive processes, but it is also important for controlling programs which may get into infinite computations. Since any code may be invoked from a competitive process, sponsors must be pervasive. Yet, since sponsorship is not an important part of specifying many procedures and algorithms, it should have a low or invisible profile in code which does not use it. This is achieved in Acore by controlling the sponsorship of ask expressions, the primary means of expressing transactions, implicitly by context. The success of the resulting invisibly low profile is evident from the lack of any sign of sponsors in the Acore code presented thus far.

The sponsor of any ask expression in the handler of a behavior defaults to the sponsor of the incoming request. Thus, subtransactions are defaultly sponsored by sponsor of the parent transaction. To specify another sponsor, the *with-sponsor* form is available:

```
(with-sponsor sponsor
  body...)
```

The forms wrapped in the *with-sponsor* form are evaluated with the actor specified as *sponsor* as their sponsor (unless, they are wrapped in a nested *with-sponsor* form, of course). By default, the incoming sponsor is bound to the identifier sponsor in the same way that the target actors is bound to self.

Using *with-sponsor*, we can optimize a parallel or by stifling sponsorship once the value has been determined, as in figure 4-16. Note that arbitrary stifling in this manner is not always safe; if the test expressions invoke transactions which are not prepared to be stifled, then stifling those transactions may

```
(let ((new-sponsor (make-simple-sponsor-script
                     sponsor  ;; parent sponsor
                     1000)))  ;; initial number of ticks
  (let ((value (with-sponsor new-sponsor
                 (or test1  ;; parallel or
                     test2
                     test3))))
    (:stifle new-sponsor)
    body using value...))
```

Figure 4-16: Using a parallel or with stifling.

produce problems. In many cases, however, the parameters to or are predicates of some sort or another which do not have side effects and may be safely stifled.

Sponsors are actors like any other, and the power of the sponsorship mechanism comes from being able to specify the behavior of the sponsors used to manage a transaction. For example, the simple sponsor created in figure 4-16 might have the behavior shown in figure 4-17.

Sponsors may have more complex behaviors, and may interact with the rest of the program in other ways. For example, a sponsor may receive progress reports of partial results from the sponsored transactions to help it decide how to focus efforts on the most productive techniques. A sponsor for a shared subtask may have several parent sponsors from which to ask for more resources, corresponding to the transactions needing the results of the subtask. These examples are just a few ideas to illustrate the wide possibilities in management structure possible with sponsors of arbitrary behavior.

To sum up: The sponsorship mechanism of Acore provides a way of controlling competitive concurrent subprocesses. Sponsorship is specified in terms of *transactions* to tie it to the flow of control within concurrent processes. Sponsorship is managed through *sponsors* who control the allocation of *ticks* charged for each transaction. An integral part of sponsorship is the stifling of transactions when sponsorship ceases, but in order to stifle transactions without leaving inconsistencies we need a clean way of handling exceptions. This is the subject of the next section.

4.9 Complaint Handling

Competitive concurrency introduces the need to stifle subprocesses, which in turn requires good exception handling to clean up uncompleted transactions. In this section I will discuss the question as to what model of exception handling Acore should follow, what is provided, and what directions are left open for further development once experience shows need for stronger capabilities.

```
(DefName Simple-Sponsor-Script
 (script (parent-sponsor tick-supply stifled?)
  ((:stifle ())                                        ;; Request to stop sponsorship
   (cond ((:> tick-supply 0)
          (:excess-sponsor-ticks parent-sponsor tick-supply)))
   (ready (tick-supply 0) (stifled? T))
   ':stifled)
  ((:more-sponsor-ticks (max-allowed))                 ;; Request from system for ticks
   (with-sponsor parent-sponsor
    (cond (stifled? (ready)
                    (complaint :sponsorship-denied))
          ((:<= max-allowed tick-supply)
           (ready (tick-supply (:- tick-supply max-allowed)))
           max-allowed)
          ((:> tick-supply 0)
           (ready (tick-supply 0))
           tick-supply)
          (else
           (let-except ((new-supply (:more-sponsor-ticks parent-sponsor 1000)))
                  (except-when
                   ((:sponsorship-denied ())
                    (ready)
                    (:stifle self)
                    (complaint :sponsorship-denied)))
            (if (:<= max-allowed new-supply)
             (then (ready (tick-supply (:- new-supply max-allowed)))
                   max-allowed)
             (else (ready (tick-supply 0))
                   new-supply)))))))
  ((:excess-sponsor-ticks (returned-ticks))            ;; System returned ticks
   (if stifled?
    (then (ready)
          (:excess-sponsor-ticks parent-sponsor returned-ticks))
    (else (ready (tick-supply (:+ tick-supply returned-ticks)))
          ':done)))))

(DefFunction Make-Simple-Sponsor (parent-sponsor initial-ticks-desired)
 (let-except ((granted-ticks
               (:more-sponsor-ticks parent-sponsor initial-ticks-desired)))
         (except-when
          ((:sponsorship-denied ())
           (complaint :sponsorship-denied)))
  (:create simple-sponsor-behavior
       parent-sponsor granted-ticks nil))))
```

Figure 4-17: A simple sponsor.

By itself, transaction stifling requires an exception handling mechanism which just deals with aborting transactions. The primary concern is to provide a way for ensuring that actors which become insensitive while calculating their replacement behaviors aren't left insensitive forever just because a transaction was stifled; they should be reverted back to their previous state so subsequent messages may be processed. Therefore a model of exception handling which deals with exceptions *only* as errors is unacceptable.

Instead, Acore uses a model of exceptions as an alternative form of response. Normal responses are *replies*; exceptional responses are *complaints*. Normal responses provide values to expressions; complaints invoke the exception handling mechanism. By default, a handler with an expression body forwards the complaint on as the response to the outer transaction. In this way stifled transactions are closed off with a complaint, and each exception handler gets a chance to perform any cleanup necessary, e.g. *ready*ing insensitive actors.

Complaint handling must be specific to the location in the program where the complaint occurs. For example, if an exception occurs while computing the replacement behavior of an insensitive actor, then the actor needs to be *ready*ed, reverting back to its previous behavior. But once the replacement behavior has been computed, *ready*ing would specify a second replacement behavior, producing an erroneous situation. Since the replacement behavior may have several parameters which are computed separately, this situation leads to a form of exception handling probably peculiar to Acore: *let-except*. *Let-except* statements have the following form:

```
(Let-except (let-bindings ...)
        (except-when
         ((complaint-keyword (complaint-parameters))
          handler-body)
         ...more complaint handlers...)
  let-except body...)
```

Let-except works just as a normal *Let* does when there are no exceptions — the expressions for the bindings are concurrently evaluated and the values are bound to the identifiers, and then the body of the *let-except* is evaluated in the environment extended by the bindings. However, if one of the expressions in the *let-bindings* produces a complaint, the complaint is routed to one of the handlers and the value of the *let-except* is the value returned by that handler; the *let-except* body is *not* evaluated.

For example, consider the bank account record in figure 4-18. Here the reason for the *let-except* syntax becomes more apparent. If an exception occurs in calculating the parameters to the replacement behavior, e.g. the new-balance and new-interest-to-date, then the *ready* in the *let-except* body cannot be performed, and the insensitive bank account must be reverted back to its previous behavior. On the other hand, if these parameters are successfully calculated, then the *ready* in the *let-except* body *can* be performed, and therefore it is an error to try to specify a second replacement behavior to revert the insensitive bank account back to its previous behavior.

```
(defname bank-account-script
 (script (balance interest-to-date)
  ...(other handlers omitted)...
  ((:add-interest (interest-amount))
   (let-except
       ((new-balance (:+ interest-amount balance))
        (new-interest-to-date
         (:+ interest-amount interest-to-date)))
       (except-when
        ((some-exception (&rest parms))
         (ready) ;; revert to previous behavior
         (complaint* some-exception parms)))
     (ready (balance new-balance)
          (interest-to-date new-interest-do-date))
     (make-interest-receipt self interest-amount
                    new-interest-to-date)))))
```

Figure 4-18: Example of *Let-Except*: adding interest to an account.

Traditional exception handling forms allow programs to wrap exception handling around an expression or a body of forms; however, to use that form here we would have to either wrap the entire *let* or we would have to wrap the expressions in the bindings. If we wrap the entire let in an exception handling form, then the exception handler will specify the previous behavior whenever there is an exception anywhere within the *let*, including the body. Thus the exception handler has no way of distinguishing whether the exception occurred in the arms or within the body, and can't tell whether it should specify a replacement behavior. If we wrap the expressions within the bindings, then the exception handling must be duplicated for each expression. For these reasons, the *let-except* form has been introduced in Acore.

However, *let-except* is not the most convenient form for use in many situations. For example, to use it in command context the programmer must pull the parameters of the command out into the arms of a *let*. As I tried to argue in the above paragraph, a form for simply wrapping other forms is often not the right thing, especially when applied to a *let* with several forms in its body. However, such a form can be useful for wrapping a single command. Complaint handling in Acore is still an area of ongoing research, so additions and changes are possible as we search for practical and elegant solutions. Some more issues are outlined below.

Complaints and *let-except* provide a way to specify complaint handling for aborted transactions, but there are other types of exceptions. For some types of exceptions there are often several ways to proceed; in different contexts, a different method of handling the exception may be appropriate. For example, in some cases a divide by zero exception inside a particular mathematical function should be

treated as an error — no zeros are expected in the input. In other situations, the zero should be treated as an approximation of an infinitesimal, and an approximation of infinity should be returned as the result of the division. Another example is dealing with outside peripherals and sensors — in some cases, exceptional conditions can just be treated as noise in the input, whereas in other cases where the information is more important a retry may be necessary.

The current exception handling mechanism makes no provision for choosing different ways of proceeding other than to write different versions of the same code or to indicate which way of proceeding is desired through an additional parameter. Complaints and complaint handling provide a mechanism for expressing exception handling in the case where the transaction must be aborted; this will serve as a starting point for experimenting with stifling transactions. Acore will have to be extended when it becomes apparent that this limitation is a problem and a clear mechanism for solving it is developed.

4.10 Top Level Naming and Modules

Acore is designed as a language for experimentation, in particular for experimenting with concurrent languages based on the actor model of computation. Since it will be used largely for exploring program design, it should provide a form which promotes incremental tinkering with programs, a form which permits incremental compiling or interpretation. In this section I discuss the nature of top level names and the issues behind their design to motivate the forms of Acore which define top level names and their scope.

4.10.1 *DefName*

To support incremental development, Acore includes a level of indirection for top level names. The *DefName* form, for example:

(*DefName* **my-account** (:create account-script ...))

associates with the name my-account a forwarding actor which forwards messages to the actor returned by the expression. This association is stored in a top level environment called the *loader table*. Any free identifiers in the expression, such as account-script in the example, are looked up in the loader table. The loader table is named such because references are looked up only when actors are loaded into the system or defined; it could be a bottleneck if it was consulted during run time.

The level of indirection supplied by the forwarding actor provides two capabilities: the ability to make forward references, and thus allowing mutually recursive definitions, and the ability to make incremental changes without relinking all the programs which use the change. For the purposes of mutually recursive definitions, it is possible to think of all the top level *DefName* definitions as making up one big LetRec like in figure 4-13. However, top level forwarding actors may also be sent a special message to change their forwarding address, so that developers can experiment with new definitions without recompiling or linking all the code which uses those definitions. For example, I could change the definition of my-account above, and recompile the new definition without having to worry about

relinking all the programs which use my-account. This can especially be a timesaver when the name is widely used in a program, and is part of what makes Lisp systems a good prototyping environment.

This capability to change forwarding is available only of top-level forwarding actors, not of the forwarding behaviors which futures take on after their value has been computed. This capability is intended only for use while developing programs, not by running programs, though certain systems functions such as interpreters may need to make use of it. Unchangeable definitions can be made by the *defequate* form, which I discuss next.

4.10.2 *DefEquate*

As discussed in section 4.6.3 there are situations where the forwarding actor is not entirely equivalent to the actor to which it forwards. Therefore on occasion it becomes necessary or desirable to define a top level name without the indirection; for this the *DefEquate* form is provided, e.g.:

```
(DefEquate pi 3.141592653589793)
```

Since no indirection is used between the name and the definition, the advantages of the indirection are lost. In particular, no forward references may be made to a name defined with *defequate*, and if the name is redefined, any code which uses it must be reloaded.

One consequence of this is that any definitions defined with *defname* may be loaded concurrently, while definitions defined with *defequate* must be loaded before any definitions which refer to them.

4.10.3 *DefModule*

While designing and developing a system of actors, many actors may be defined at top level, but only some of them should be accessible outside the system. This especially becomes a problem for guaranteeing small interfaces between modules in developing large systems, but is also a problem for hiding names in modules generated by macros.

Acore already includes two forms which may be used to encapsulate names in a system into a local scope: *let* and *script*. These can be used in conjunction with *defname* to define a system of actors with hidden names inside it. Mutually recursive definitions can be defined using the LetRec of figure 4-13, so for example a quicksort module could be defined as:

```
(defname quicksort
 (letrec ((quicksort ...)
          (partition ...))
  quicksort))
```

However, these forms fall short of addressing the problem of top level names by making it awkward to define systems with more than one interface, i.e. to express systems which define more than one name at top level.

To address this problem, Acore provides the *DefModule* form for limiting the scope of top level

names. *DefModule* has the following syntax:

(*DefModule module-name* (*exported names...*)
definitions...)

and is used as follows:

(*DefModule* **quicksort-module** (quicksort)
(*DefName* quicksort ...)
(*DefName* partition ...))

The module definition consists of the module name, a list of exported names, and the definitions. The module name is currently ignored; in the future it may be used to organize libraries or locate source code. The list of exported names are the top level names defined within the module which are visible in the context outside the module. The advantage of *defmodule* is that this may be a list, so that several names may be exported. The names defined in the body of the *defmodule* are visible only by other definitions within the body; unless they are exported, they are not visible outside the module. Thus, in the example above, only quicksort is visible outside the module, while partition is only visible inside the module.

Top level naming in Acore is designed to permit incremental development and modification of systems, necessary for promoting fast prototyping of experimental systems. *DefModule* provides a facility for encapsulating names internal to systems so the interface to modules which have been developed can be restricted. This encapsulation of names also applies to names of syntactic abstractions introduced with the macro facility, which is described next.

4.11 The Macro Facility

The final important aspect of the design of Acore I will discuss in this chapter is the macro facility. Acore is designed as a foundation for experimentation with the design a class of concurrent languages, so it is important to be able make syntactic as well as behavioral abstractions.

The uniform, fully parenthesized syntax of Acore is well suited for defining extensions to the syntax of the language. Since the parsing of all forms is defined in advance, adding syntactic extensions to the language which conform to this is very easy; Lisp programmers use this to advantage. However, the macro expansion facility of Lisp is not entirely suited for experimenting with, extending, and changing the semantics of a language. In particular, the syntax of special forms is coded into the macroexpander, so it is difficult to specify special processing for syntactic extensions (basically it requires redefining the macroexpander). Also, because of this special processing it doesn't provide a clean method for redefining special forms, nor does it provide a simple hook for redefining the semantics of identifier lookup or simple applications. Therefore, another approach is required.

The approach taken in Acore is largely based on the "expansion passing style" proposed in [Dybvig et al. 84], so I won't gone into great detail here. I will outline this macro mechanism, point out

its strengths, and explain the extensions to it made in Acore.

4.11.1 "Traditional" Macros

The basic strategy of traditional (defmacro style) Lisp macros is to recursively walk over a form, copying it and replacing any macro form, a form which starts with a symbol which has a macro definition, with its macroexpansion obtained by applying the macro function to the form. If any macroexpansions were made, then once this is done the macroexpander starts over and does it again, until no more macroexpansions are made, indicating that no macro forms remain. It avoids macro expanding special syntactic forms such as the parameter list of a lambda by recognizing processing special forms and processing them specially.

4.11.2 Expansion Passing Style Macros

The basic strategy of expansion passing style macros is to apply a general *macroexpansion function* to a form once. The macro expansion function looks at the form and calls the appropriate *expander function* for that form, whether it has a macro definition, or is a simple application, or is an identifier. It does not attempt to expand subforms; that is the job of the expander for that form. It passes another argument to the expander function, the macroexpansion function to be used to expand the subforms.

The form specific *expander function* may process the form any way it wants, but it is responsible for ensuring that the form that it returns is valid code without any macros which need expansion. In most cases this requires at least applying the macroexpansion function to the subforms; this is what the default expander for applications does. Sometimes it is sufficient just to return the form as it stands; this is what the default expander for identifiers does.

All capabilities of traditional style macros can be expressed using expansion style macros. As the authors show in their paper [Dybvig et al. 84], defmacro style macros can be defined by defining expanders which call the passed macroexpansion function on the entire form returned. The idea is that

```
(defmacro name args
  body)
```

becomes

```
(defexpander name (form expander)
  (expander (apply (lambda args body) form) expander))
```

though this could be done in a way so the destructuring of *form* is performed explicitly rather than by using apply. Also, performing one step of the expansion for debugging can be accomplished simply be passing a macroexpansion function which does no further expansion, e.g.

```
(defun expand-once (form)
  (initial-expander form (lambda (subform expander) subform)))
```

The advantages of the expansion passing style arise when defining new forms with special syntax

and for redefining language forms. Special forms in the language must have expanders defined which take care of expanding any subforms which need expanding; for example, the expander for lambda should expand the lambda body but leave the parameter list untouched. Since new expanders also have complete control over which subexpressions are expanded, defining new special syntax is no problem. Since the form returned by an expander is not re-expanded, it is possible to define macros which change the semantics of language forms by redefining the new form in terms of the old form; with traditional macroexpansion defining a macro in terms of itself always results in an infinite loop.

Using the ability to redefine the application expander and the identifier expander, it is possible to radically redefine the semantics. For example, in the paper [Dybvig et al. 84] the authors demonstrate how to give Lisp a call by name semantics using this technique.

4.11.3 Acore Macros

Acore macros provide the same capabilities as the expansion passing style macros. However, I have modified the implementation to correct the scoping of macro names according to the syntax of Acore.

In the traditional implementation of macros as well as the expansion passing style implementation, macros are stored as global properties of the symbols. However, in Acore, top level names may have limited scope within modules; therefore a global table is not the best implementation. Instead, Acore stores macro definitions in a macro environment, and frames are pushed on this environment upon entering modules. The environment is passed as an additional argument to the expander functions, which can then use it to look up macros. Macro definitions are just macros which store an expander function in the environment.

One advantage of expansion passing style perhaps missed by the authors of the paper is that it can be used to protect locally bound names from erroneous macro expansion. Locally bound names, such as the parameters to a lambda or the identifiers bound in a *let*, should not be subject to macroexpansion within the scope of their binding. For example, the following code should not change meaning just because someone later decides to name a macro par:

```
(lambda (par)
  (par x y))
```

The technique used by the Acore macro facility is to define the macro expander for forms which bind names (e.g. lambda) in such a way that within the scope of those names (e.g. the body of the lambda) any forms beginning with the bound name are treated as normal applications expanded by the application expander. One way to do this is to locally define the name as a macro whose macro expander is the application macro expander; this guarantees that it will be treated as an application. Thus, in the above example, within the body of the lambda, par is defined to be macro with an expander which

behaves identically to the application expander.[4]

Two predefined macros serve as the means of defining expanders.

(*DefExpander name (macro-environment next-expander form) body...*)
(*DefMacro name (macro-parameters...) body...*)

DefExpander is the general form for defining expansion passing style macros. *Macro-environment* is the environment described above; *next-expander* is the macroexpansion function for further expansion; *form* is the source form to be expanded. *DefMacro* is a simpler form which can be used for defining macros in the traditional style; we saw how this can be done using expansion passing macros. Currently the Acore compiler is written in Lisp, so the macro bodies are Lisp forms for dealing with the source code in terms of Lisp list structure. Two examples of Acore macros are shown in figure 4-19.

```
(DefMacro Lambda (parameters &rest body)
  '(:create (script ()
              ((:do ,parameters :unserialized)
                ,@body))))

(DefExpander With-Futures (env expander form)
  ;; wraps (future ...) around every ask expression in form
  (let ((local-env
         (install-expander
           ':application-expander
           (lambda (env expander form)
             '(future ,(map-expander expander env form)))
           (push-macro-environment-frame env)))
        (body (cdr form)))
    (if (cdr body)  ;; if multiple forms in body, wrap (let ()...) around them
        '(let () ,@(map-expander expander local-env body))
        (expander local-env expander (car body)))))
```

Figure 4-19: Example expanders: Lambda, With-Futures

The lambda macro defines lambda in terms of Acore primitives; it expands into a form which creates an instance of an anonymous unserialized function behavior. With-Futures is a macro which wraps around a body; it redefines the application expander within the context of the body so that all applications within the body are evaluated eagerly.

Acore macros thus provide a flexible facility for introducing syntactic abstractions into Acore, including abstractions which change the semantics of the existing language. The ability to change the semantics of applications (ask expressions) is especially interesting, and I'm sure people will try using it

[4]Actually, since looking up macros is faster than building a new macro frame environment in the current implementation, the macroexpander only shadows definitions for names which actually do conflict with a macro name.

with *delay* and *future* to experiment with delayed and eager evaluation strategies.

4.12 Summary

Acore has been designed with several major goals in mind. In this chapter some of the issues concerning designing a language with those goals have been raised, and the solutions provided by the design of Acore have been discussed. Some of those issues and solutions are:

- Procedures using cooperative concurrency in a divide and conquer manner are easily expressed by the expression oriented syntax of Acore.
- *Script* expressions express behavioral abstraction, combining the closures of function oriented languages with the message passing interface of object oriented ones.
- *Referential transparency* of identifiers in Acore keeps the semantics of the expression oriented syntax simple even with concurrent evaluation.
- The actor model of state change through replacement behavior preserves referential transparency of identifiers despite state change since a new set of identifiers is bound for each message.
- Commands and command context provides the ability to express behaviors which do not return values.
- *Futures* provide a level of indirection which can be used to implement future concurrency, delayed evaluation, and mutually recursive definitions. However, this level of indirection doesn't come for free; forwarding actors are distinguishable from their forwardees in contexts where not only are messages sent to references, but references themselves are compared.
- The *race* expression provides a way of expressing competitive concurrency, returning a queue of the values in the order returned. It is not completely general, but its form is highly useful for expressing common static competitions through macros. Explicit queues can be used for more dynamic situations.
- Acore meets the problem of controlling competitive subprocesses by controlling the processing used by *transactions* through a *sponsorship* metaphor. Sponsors are actors who may have arbitrary behaviors, permitting many possible management structures.
- *Stifling* unneeded transactions brings up the problem of cleanly aborting transactions. Acore addresses this point by returning *complaints* in place of values for stifled transactions, and providing the means of providing cleanup processing upon receiving a complaint through *let-except*. However, this mechanism will probably need to be extended to provide full exception handling.
- Acore provides a programming environment for incremental development through indirection of top level names. It also provides a mechanism for narrowing the visible interface of modules by controlling the scope of their top level names with *defmodule*.
- Finally, Acore provides a flexible macro facility which allows redefining the semantics of existing forms, including semantics of application (ask expressions) and identifiers, increasing the opportunities for language experimentation. The macro facility also preserves the lexical scoping of local names.

With the design of Acore in mind, we can now take a look how it may be implemented and compiled.

Part III
Compiling Acore

Chapter Five

Introduction to Implementation

We now turn to the implementation of Acore, in particular the implementation of its compiler. This chapter presents a brief overview of the implementation so the reader may see how the pieces fit together, both in terms of how Acore fits into an implementation and how the parts of the compiler fit together.

Acore is part of the Apiary Project, a system for experimenting with concurrent actor programs. Acore is supported by two lower levels of the project, as illustrated in figure 5-1.

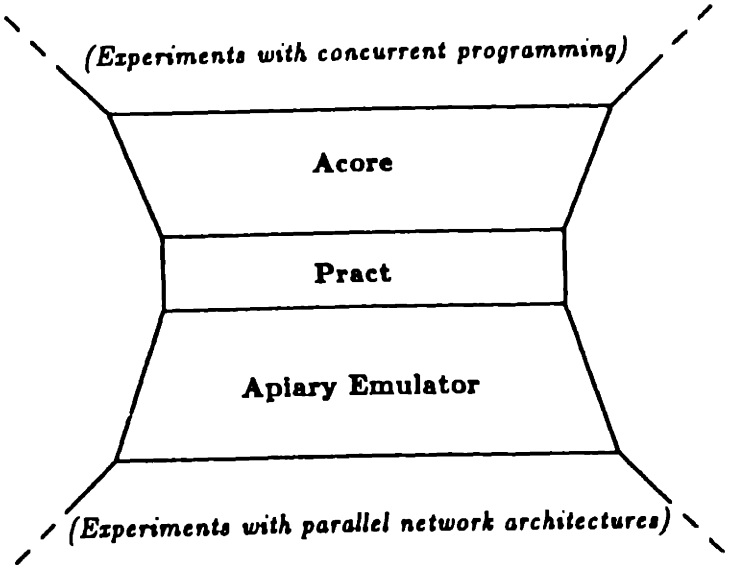


Figure 5-1: Organization of the Apiary Project

The *Apiary* is a design for an architecture for running actor programs on a closely coupled network of processors. The *Apiary Emulator*, while not critical to an understanding of the Acore compiler, is described in Appendix A to give interested readers a feel for some of the issues involved in implementing actor languages. The emulator is designed primarily to allow people to run and experiment with compiled Acore programs, so it includes hooks for observing and debugging features. *Traveler*, the Apiary observatory, uses these hooks to attack the problems of observing and debugging concurrent actor programs, and is described in Appendix B.

Pract, a *pr*imitive *act*or language, serves as the language for describing actors at the primitive

level. This is the level at which actors may be implemented in a computer architecture, and it is the level at which actors are dealt with theoretically [Agha 86]. It serves as the interface between actor languages such as Acore and the architecture for running actor programs, in this case the Apiary Emulator. It is the target language for the Acore compiler, so since an understanding of it and the primitive actors it describes will help define the task of the Acore compiler, the next chapter is devoted to discussing the capabilities of actors at this level and how behaviors described in Acore may be expressed using Pract.

The Acore compiler translates scripts written in Acore into sets of scripts written in Pract, organized to implement the same behavior. The major duty of the compiler is to separate the concurrent behavior described with ask expressions in an Acore script into a system primitive behaviors devoid of ask expressions. Since primitive actors process message using only their local state, another major aspect of the compiler is to keep track of references during compilation so that the compiler can organize them to make sure that each primitive actor generated has the references it needs to perform its part at run time. After examining the question of *what* the compiler should produce in the next chapter, we explain *how* the compiler performs this transformation in Chapter 7.

Chapter Six

Compiling to Primitive Actors

The actor model of computation provides a suitable model for implementation on a message passing parallel computer architecture. Since an actor may process a message using only its the local state (including its behavior), machines which process messages for primitive actors needn't block waiting for distant memory accesses. This also allows primitive actors to be easily migrated from machine to machine to balance processor and memory loading. In this chapter we will outline the abilities of these primitive actors and see how the variety of behaviors in Acore may be implemented within the limited behavioral capabilities of primitive actors through organization. This chapter is concerned with the issues of *what* the Acore compiler should produce; the following chapter looks into *how* it may do so.

6.1 Primitive Actors

As described in Gul Agha's book [Agha 86], primitive actors have the limited capabilities to make simple decisions, create new actors, send new messages, and specify a replacement behavior. All the actions of the actor are a function of its current behavior, its local state, and the incoming message. Primitive actor behaviors have much in common with behaviors in Acore. Their behaviors are expressed in scripts, which are divided into handlers for processing different kinds of messages. The behavior of a primitive actor is parameterized by the acquaintances stored in its local state.

However, one of the major contributions of Acore over the expression of behaviors directly in terms of primitive actor behaviors is the introduction of ask expressions. Ask expressions are fundamentally outside the scope of what can be expressed directly as part of a single primitive actor's behavior, since an ask expression implies sending a request message and *waiting* for a reply. A primitive actor's behavior defines its reaction to an incoming message, an event. Yet an ask expression involves two events: the incoming event which starts the subtransaction, causing the actor to evaluate the ask expression in the first place; and later the reception of the reply. Thus ask expressions must involve two behaviors, one which initiates the request and one which later receives the reply.

To express behaviors at this primitive level, I have designed the primitive actor language *Pract*. Since behaviors at the primitive level are similar to behaviors at the Acore level in many respects, the syntax of the two languages is very similar. Pract is also similar to the languages presented in the Agha's book, but has been enhanced in several respects, mostly for practical considerations.

To illustrate some of the differences, let's take a look at part of an example we've seen before, the locker from Chapter 2, but this time written in Pract. See figure 6-1. This is not how the Acore compiler

would compile the example, for reasons we will see below, but serves to point out differences of Pract. (You may also find the *Pract Reference Manual* in Appendix D helpful for more details about Pract.)

```
(defequate key-behavior (script (() ())))

(defequate empty-locker-behavior
 (script
  (() (full-locker-behavior key-behavior)
   (is-request ((:store valuables))
    (let ((new-key (create key-behavior)))
     (replace self full-locker-behavior new-key valuables)
     (reply-to customer (reply-keyword my-key))))
   (is-request ((:retrieve key) :unserialized)
    (complain-to customer (:wrong-locker reply-keyword))))
  full-locker-behavior key-behavior))

(defequate full-locker-behavior
 (script
  ((my-key valuables) (empty-locker-behavior)
   (is-request ((:store new-valuables) :unserialized)
    (complain-to customer (:full-locker reply-keyword)))
   (is-request ((:retrieve customer-key))
    (if (== customer-key my-key)
     (then (replace self empty-locker-behavior)
         (reply-to customer (reply-keyword valuables)))
     (else (update self)
         (complain-to customer (:wrong-key reply-keyword))))))
  empty-locker-behavior))
```

Figure 6-1: A simple locker written in Pract (nonoperational — see text).

6.1.1 Actor Creation

We've said that actors must have the ability to create new actors, but before now all actors have been created by sending a :create message to the script guardian. One reason for this was mentioned in Chapter 4: this guards against erroneously creating actors with the wrong number of acquaintances. Another reason for this follows from the fact that actors can respond to messages using only their local state and behavior: an actor is malformed if it is created with a forwarding actor to its script rather than an actual script. Since several constructs in Acore introduce forwarding actors, e.g. *defname* and *future*, scripts can be handled uniformly like other actors only if the language provides a message passing interface for creating actors and replacing behaviors with the script. The guardian for the script provides this interface.

In light of this, the first thing to notice about figure 6-1 is that the Pract *create* expression is being used to create actors. *Create* is the primitive form for creating new actors; it does not require any message passing. Since there is no communication with the script or any of the actors involved, it cannot

perform any checking, but simply creates an actor from the parameters, the script and the initial acquaintance values.

Since *create* must use the script and not a forwarding actor to the script, the scripts have been defined using *defequate* rather than *defname*. This points out why this translation of the previous example is nonoperational — since there is no indirection using *defequate*, recursive *defequate*'s can not work. Thus, another reason for encapsulating scripts in guardians is to permit a level of indirection so that recursive script definitions are possible.

We conclude from this that although actors at the Pract level must have the primitive capability to create actors, at the Acore level actor creation should be encapsulated in script guardians for three reasons: to abstract creation of actors so that parameter checking may be performed, to keep the interface to actors uniform despite forwarding actors and futures, and to permit recursive script definitions. Therefore, since actor creation in Acore is encapsulated in a guardian for the script, the guardian encapsulates a *create* known to be correct, and actor creation is just another ask expression in Acore programs. We will explain how ask expressions are implemented soon, but first there are a few more differences to extract from this example.

6.1.2 Message Passing

Actors have the ability to send messages. In Pract, all message passing is explicitly expressed with *request*, *reply-to*, and *complain-to* commands. In addition, all the components of the messages are explicit as well. Thus, as is apparent in figure 6-1, *reply-to* and *complain-to* commands must not only include any values returned, but also the *customer*, the actor to which the returning values are being sent, and the *reply keyword* for selecting a handler. The *request* command includes not only the target, selector keyword, and message parameters, but also a sponsor for the transaction, a customer to whom to send the reply, and a reply keyword with which to send the reply (see example *request* in figure 6-2). Similarly, handlers for messages are specified with *is-request*, *is-reply*, and *is-complaint*, which bind the parts of the message. However, the common parts of these bindings are implicitly bound to self, sponsor, customer, and reply-keyword. (These bindings may be overridden — see the *Pract Reference Manual* included in Appendix D for more details).

The actor model specifies only that actors be able to send messages, not that they have any intrinsic type. In Pract, however, I have separated normal messages into three basic types: requests, replies, and complaints. There are several reasons for distinguishing requests from responses. First, requests have several required parameters which are not part of response messages: the sponsor, the customer, and the keyword with which to send the reply. Separating the types allows the Pract compiler to check that these are supplied as necessary for requests. Second, this separation makes things easier for debuggers, since they may display messages according to their function. For example, requests can be paired with their corresponding reply. Third, an implementation may be able to optimize some of the choices about handling these different kinds of messages since the choices are moved to compile time.

Update and replace are also messages; they are special messages involved in specifying replacement behavior and will be discussed later.

6.1.3 Scripts

In the actor model, behaviors are black boxes, perhaps created by parameterizing a behavior expression with values for acquaintances. In Pract, behaviors have been separated into three components: the *script*; the set of *local acquaintances*; and the set of *constant acquaintances*. The *script* specifies control flow by means of its handlers. The *local acquaintances* specify the references which are private to a particular instance of a behavior. They are separated from the script because behaviors frequently change only by updating the value of the local instance acquaintances, so by separating them from the rest of the behavior it is possible to optimize this case. The *constant acquaintances* specify the references which are constant and shared by all instances of the script. The Pract script expression differs from the Acore script expression by declaring the names of the free identifiers and their values explicitly; note the declaration of (empty-locker-behavior) in the full-locker script of figure 6-1, and how its value must be specified at the end of the script expression. (See the Pract manual included in Appendix D for syntactic details). This is primarily for compilation efficiency: with all identifiers declared up front, Pract can be implemented as a one pass compiler.

6.1.4 Primitive Expressions

The actor model does not specify which expressions should be primitive, requiring no message passing. Besides actor creation, the only primitive expression necessary is one which distinguishes references for primitive decision making. In Pract the identity (==) expression determines whether two references are identical or not; for example it is used to tell whether the identical key was passed for the locker in figure 6-1.

An *if* expression has also been included in Pract, since it greatly simplifies the compilation of simple *if* expressions in Acore. For example, see figure 6-2. Not only does this solution save duplication of code, but the transformation required can become very complex with *if*s arbitrarily nested. I also chose to include the capability to combine boolean values with or, and, and not in Pract.

Arithmetic operators could have been included in Pract, and in fact an efficient implementation may need to provide in-line operations on integers, but I chose not to include numerical operations in Pract. Unlike boolean values, users *are* likely to develop further types of numbers and number-like objects and they can best share code (e.g. for sorting) if all interactions with numbers are through a message passing interface.

With this quick overview of the restrictions on primitive actors and the differences between Pract and Acore this causes, we now turn to questions of how Acore behaviors can be implemented in terms of these primitive behaviors, starting with the ubiquitous ask expression.

The Acore expression:

```
(let ((y-or-n (if (== x y) ':yes ':no)))
  (:print stream y-or-n)
  (ready (answer y-or-n))
  y-or-n)
```

The equivalent Pract with if expressions:

```
(let ((y-or-n (if (== x y) ':yes ':no)))
  (request stream (:print y-or-n) sponsor null-customer :value)
  (update (1 y-or-n))
  (reply-to customer (reply-keyword y-or-n)))
```

The equivalent Pract without if expressions:

```
(if (== x y)
  (then (let ((y-or-n ':yes))
          (request stream (:print y-or-n) sponsor null-customer :value)
          (update (1 y-or-n))
          (reply-to customer (reply-keyword y-or-n))))
  (then (let ((y-or-n ':no))
          (request stream (:print y-or-n) sponsor null-customer :value)
          (update (1 y-or-n))
          (reply-to customer (reply-keyword y-or-n)))))
```

Figure 6-2: Demonstration of need for Pract *if* expressions

6.2 Ask Expressions

A common pattern of communication is to make a request to an actor, and later continue processing using the value returned in the reply. This is the transaction expressed by an Acore ask expression. We look now into how ask expressions can be implemented in terms of the primitive actors of Pract, choosing the method which applies most generally in the concurrent message passing world of actors.

To use the locker actor we defined above, we would make a request to store something, and then we would need to wait for the key to be returned before we can open it later (figure 6-3). Similarly, if we make a request to a function to compute a value, we can't use the value until it is returned.

For this type of transaction, the request message must contain a *customer* to whom the reply should be sent. Note, however, that if the request is recursive or if many requests may be serviced concurrently, then it won't suffice for the same actor which receives a request to be the customer for its own subtransaction. The actor which serves as the customer must hold the information needed to continue processing; however, in a recursive request, there will be several different sets of information, one for each

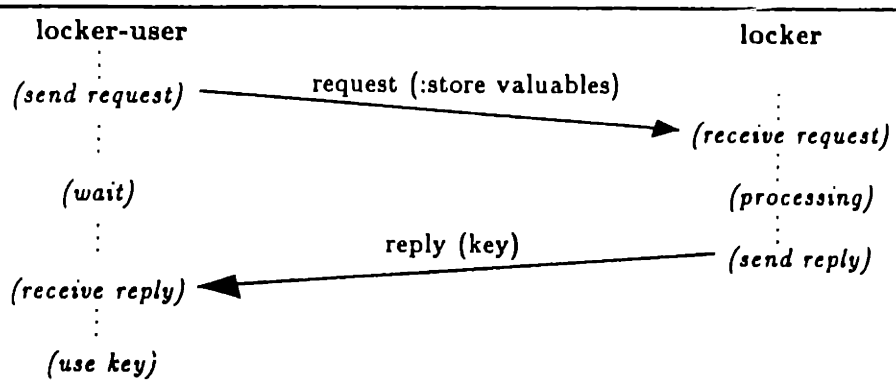


Figure 6-3: The transaction represented by (:store my-locker my-valuables).

recursive invocation. Therefore we create a *continuation* actor to store the information and continue processing once the reply is received. This continuation actor serves as the customer of the request.

For example, consider the factorial behavior of figure 6-4, written in Pract. To simplify the example, the numeric operations are performed using in-line arithmetic operations. Normally, arithmetic operations are performed by making requests to the numbers involved and setting up additional continuations to process the values returned.

The inline-factorial actor is created with the factorial-behavior. The factorial-behavior implements only the first step of the factorial; the rest of computation is performed by the continuation it creates for each transaction. Sometimes the actor which receives the initial request is known as the *leading actor* to distinguish it from the *continuations*; the *leading behavior* is similarly distinguished from the *continuation behaviors*. For clarity I have listed the leading behavior followed by the continuation behaviors in the order they are used to process a transaction, even though this produces invalid Pract code (*defequates* can't be referenced before they are defined). Reversing the order of behavior definitions will fix the problem.

Each recursive invocation of *inline-factorial* builds a new continuation actor to separately hold the information for that invocation. The continuation actors form a chain which acts like a stack of contexts. When the value is returned, each continuation does a multiplication with the value of n for that invocation and returns the result to the customer for that invocation, unwinding the stack of continuations (figure 6-5).

One advantage of the chain of continuations over a normal processor stack is that as messages are sent to different actors distributed on separate processors, the chain may cross processor boundaries. Another advantage is that since the chain is composed of actors, parts of the chain may be migrated to other processors as part of balancing the load across processors. A third advantage is that the stack may fork for concurrent subtransactions; this will become apparent when we consider concurrent transactions,

```
(defequate inline-factorial-behavior
 (script
  (() (inline-factorial-continuation-behavior)
   (is-request ((:do n) :unserialized)
    (if (in-lisp (= n 0))
     (then (reply-to customer (reply-keyword 1)))
     (else
      (let ((continuation
           (create inline-factorial-continuation-behavior
                n customer reply-keyword)))
       (request self (:do (in-lisp (- n 1)))
          sponsor continuation :value))))))
  inline-factorial-continuation-behavior))

(defequate inline-factorial-continuation-behavior
 (script
  ((n customer reply-keyword) ()
   (is-reply ((:value n-minus-1-factorial) :unserialized)
    (reply-to customer
     (reply-keyword
      (in-lisp (* n n-minus-1-factorial)))))))))

(defname inline-factorial (create inline-factorial-behavior))
```

Figure 6-4: A sequential recursive factorial using continuations and inline arithmetic.

but first we should look at sequential transactions.

6.3 Sequential Ask Expressions

Once we know how to generate continuations for ask expressions, are all our problems solved? Not really. Although in some situations a new continuation actor could be created for every ask expression, this proves to be inefficient due to the overhead of allocating and initializing the continuations. In this section we look at a particular situation where the optimization of reusing the continuation actor can be made, and some of the subtler issues which arise as a result of this optimization.

One way to combine ask expressions is to nest them, or otherwise make one transaction dependent upon a previous transaction (figure 6-6). For example, in the expression

(:deposit savings-account (:withdrawal checking-account 100))

the deposit is dependent upon the value returned by the withdrawal. For the first ask expression we can build a continuation as we did in the factorial example above. The second ask expression requires a new continuation behavior, but it may be practical to use the same continuation actor and just update its behavior.

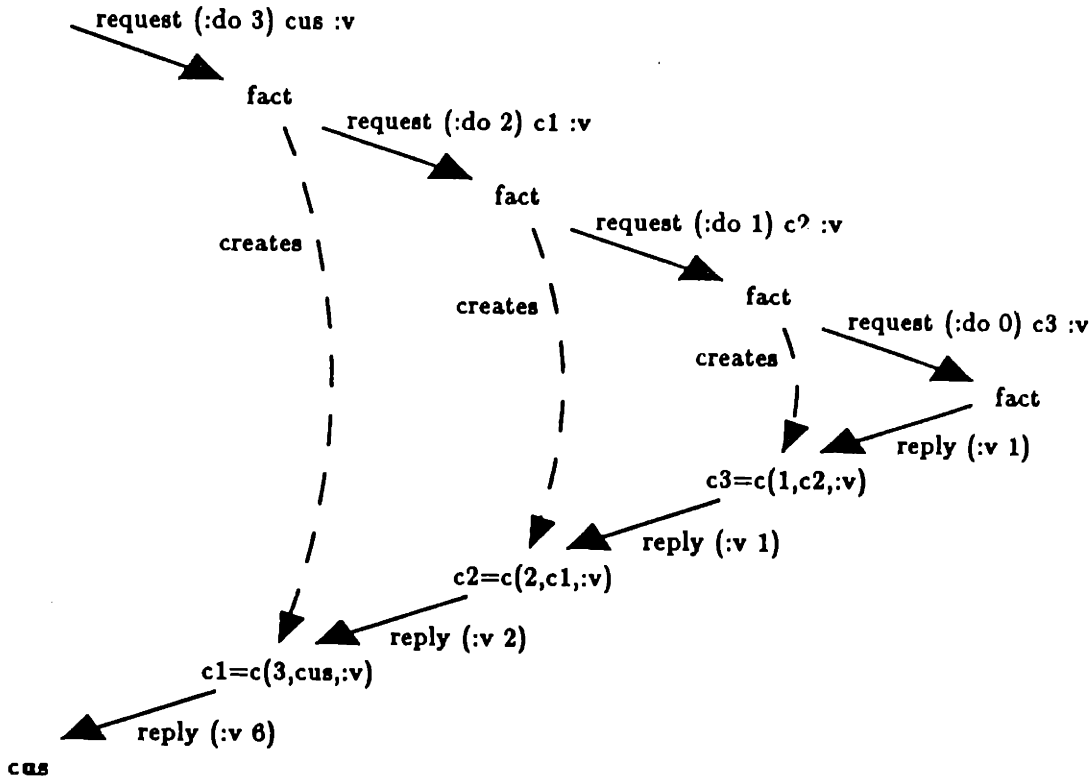


Figure 6-5: The factorial transaction: continuations form a stack.

Dependence on a subexpression:

```
(lambda (amount savings-account checking-account)
  (:deposit savings-account (:withdrawal checking-account amount)))
```

Dependence on a let binding:

```
(lambda (square)
  (let ((side-length (:side-length square)))
    (:* side-length side-length)))
```

Dependence on a condition:

```
(lambda (sum amount)
  (if (:< amount 0)
    (then (:- sum amount))
    (else (:+ sum amount))))
```

Figure 6-6: Examples of dependent ask expressions.

For example, see the recursive factorial rewritten with message passing arithmetic in figure 6-7.

This recursive factorial behavior in Acore:

```
(defequate factorial-behavior
 (script ()
  ((:do (n) :self factorial :unserialized)
   (if (:= n 0)
    (then 1)
    (else (:* n (factorial (:- n 1))))))))
```

May be implemented with the following Pract scripts:

```
(defequate factorial-behavior
 (script
  (() (cont1)
   (is-request ((:do n) :self factorial :unserialized)
    (let ((cont (create cont1 n factorial                    ; Continuation is created here
                 sponsor customer reply-keyword)))
     (request n (:= 0) sponsor cont ':v1)))))
  factorial-continuation-behavior-1))

(defequate factorial-continuation-behavior-1
 (script
  ((n factorial sponsor customer reply-keyword) (cont2)
   (is-reply (:v1 v1)
    (if v1 (then (reply-to customer (reply-keyword 1)))
           (else (update self (0 cont2))                     ; and updated here
                 (request n (:- 1) sponsor self ':v2)))))
  factorial-continuation-behavior-2))

(defequate factorial-continuation-behavior-2
 (script
  ((n factorial sponsor customer reply-keyword) (cont3)
   (is-reply (:v2 v2)
    (update self (0 cont3) (2 'ignore))          ; and here; 'ignore fills unneeded acq slot
    (request factorial (:do v2) sponsor self ':v3)))
  factorial-continuation-behavior-3))

(defequate factorial-continuation-behavior-3
 (script
  ((n ignore sponsor customer reply-keyword) (cont4)
   (is-reply (:v3 v3)
    (update self (0 cont4) (1 'ignore) (3 'ignore))          ; and finally here
    (request n (:* v3) sponsor self ':v4)))
  factorial-continuation-behavior-4))

(defequate factorial-continuation-behavior-4
 (script
  ((ignore ignore ignore customer reply-keyword) ()
   (is-reply (:v4 v4)
    (reply-to customer (reply-keyword v4))))))
```

Figure 6-7: A sequential recursive factorial using continuations and message passing arithmetic.

There are several things to notice in this example. First, only one continuation actor is created; as subtransactions are completed and the transaction progresses, the continuation updates its behavior to take on each of the continuation behaviors, and specifies itself as the customer for the next transaction (figure 6-8). The Pract *update* command specifies the replacement behavior in each case. It is like an Acore *ready* command except that acquaintances are referred to by position rather than by name, and the script may also be changed (it is acquaintance 0).

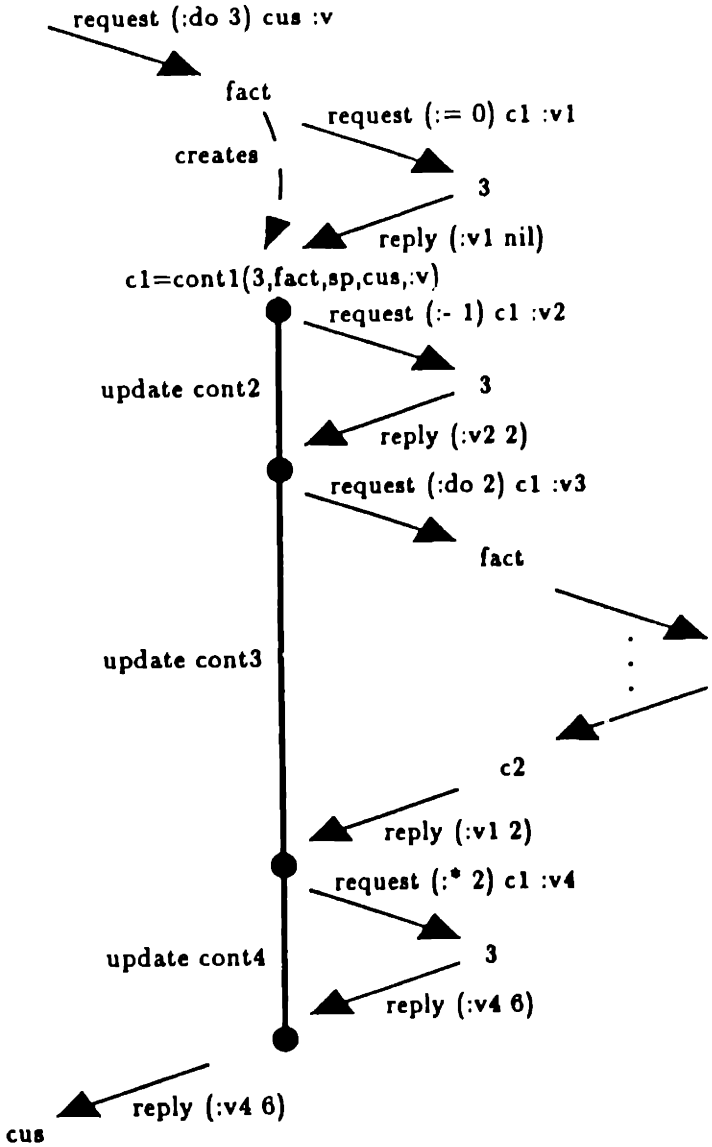


Figure 6-8: Graphical depiction of factorial transaction.

Second, as the continuation evolves through the behaviors, at different stages in the transaction it may need to keep track of different sets of values in its acquaintances. However, for efficiency purposes,

the size of the actor is not changed; instead the unused slots are merely filled with some value to be ignored. Removing a reference to an actor as soon as it is no longer needed may help improve system performance if it allows the actor to be garbage collected earlier.

Third, each subtransaction uses a different reply keyword. One hazard of using the same actor as the continuation is that there may be an error in a subtransaction which produces two reply messages; if the continuation had no way of distinguishing them, then the continuation may falsely accept the duplicate reply as the response to the second subtransaction (figure 6-9).

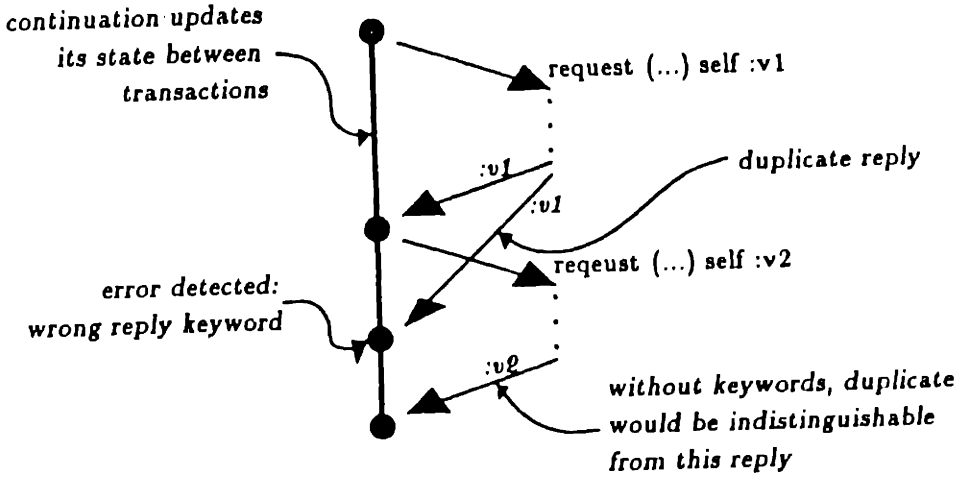


Figure 6-9: Need to catch duplicate reply error.

Fourth, a tail recursion optimization can be made. Notice that the final behavior merely forwards the result on to the customer. This behavior can be eliminated by performing the last subtransaction with the outer transaction's customer and reply keyword (figure 6-10). To make this optimization, the previous behavior is simply changed to make the last request using the outer transaction's customer and reply-keyword, and the last behavior is eliminated:

```
(defequate factorial-continuation-behavior-3
 (script
  ((n ignore sponsor customer reply-keyword) ()
   (is-reply (:v3 v3)
    (request n (:* v3) sponsor customer reply-keyword)))))
```

(Compare with figure 6-7.) For tail recursive functions, this may allow the previous invocation's continuation actor to be garbage collected before the following invocation's continuation actor is created, so that there is no net increase in space allocated.

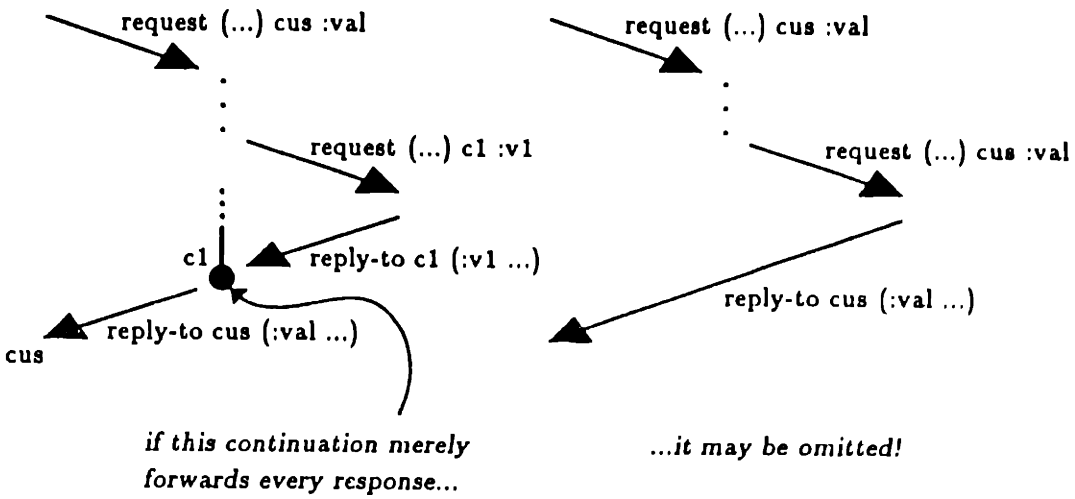


Figure 6-10: Tail Recursion Optimization

6.4 Concurrent Ask Expressions

The recursive factorial example above is basically sequential, since there are no concurrent ask expressions. We now look at how concurrent joining transactions can be implemented, and discover another need for reply keywords.

To perform concurrent transactions, two requests must be sent concurrently. If the results of the two transactions are used independently, then they may have separate continuations. They may be compiled as two sequential processes which start from the same actor, forking but never joining again. On the other hand, if the results must be used together, then the transactions must return results to a *joining continuation* (figure 6-11).

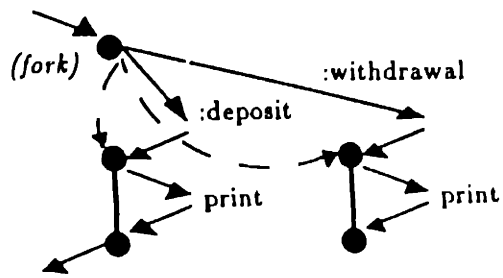
The joining continuation must perform some synchronization, since the replies may return in any order. Basically, the joining continuation must store all the replies it receives until the last one, keeping track of which reply is which so that the right values will be bound to the right identifiers. When the last of the values returns, then the joining continuation may continue.

For example, consider the inline-rangeproduct in figure 6-12. Again, inline arithmetic is used to simplify the example and exemplify the concurrent transactions. Rangeproduct is an actor which concurrently computes the product of a range of numbers, and can be used to calculate a factorial concurrently.

This example illustrates one method of implementing the synchronization of the joining customer. The joining customer is created not only with acquaintances to hold the values it has and needs later, but also with acquaintances to store the values returned by the concurrent transactions (e.g v1 and v2). In

Independent concurrent transactions:

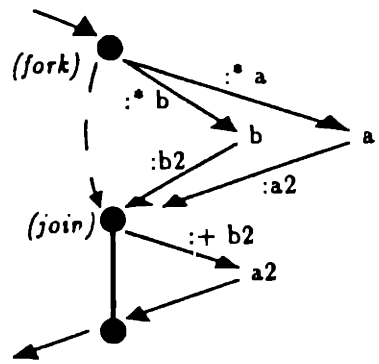
```
(lambda (account amount)
  (print (:withdrawal account amount))
  (print (:deposit account amount)))
```



No join, separate continuations

Joining concurrent transactions:

```
(lambda (a b)
  (let ((a2 (:* a a))
        (b2 (:* b b)))
    (:+ a2 b2)))
```



Join, share joining continuation

Figure 6-11: Concurrent Transactions: forking and joining

each of these slots is stored a unique value which cannot be returned as the value of a transaction; a newly created actor (unique-value) serves nicely. This value is used to determine whether a value has been returned for each transaction, so one extra slot is needed to store a reference to the unique value which won't be overwritten (the continuation's acquaintance called unique-value).

For each reply the joining customer receives, it first checks to see whether any other transactions are outstanding. If any of the acquaintance slots reserved for the other returning values still hold the unique value, then the value for the corresponding transaction hasn't been returned yet and the joining customer may not start the next transaction. Therefore it stores the reply value received in its corresponding acquaintance slot and waits for the next reply. Since the rangeproduct continuation only expects to receive two values, the check amounts to simply testing whether the other value has been received yet. In general this method checks the other $n-1$ returning value slots, where n is the number of values expected. When the last value is received, then all the other acquaintance slots reserved for returning values will be filled with values other than the unique value. At this time the continuation may

This inline-rangeproduct behavior in Acore:

```
(DefFunction Inline-RangeProduct (lo hi)
  ;; #I(f ...) means function f is to be performed inline w/out message passing
  (cond* (#I(= lo hi)
            lo)
         (#I(= lo #I(+ hi 1))
          #I(* lo hi))
         (else
          (let ((average #I(floor #I(+ lo hi) 2)))
            #I(* (rangeproduct lo average)
                 (rangeproduct #I(+ 1 average) hi))))))
```

May be implemented in Pract using the following scripts:

```
(defequate null-script (script (() ())))

(defequate inline-rangeproduct-behavior
  (script
   (() (cont1)
    (is-roquest ((:do lo hi) :unserialized)
     (if (in-lisp (= lo hi))
       (then (reply-to customer (reply-keyword lo)))
       (else
        (if (in-lisp (= lo (in-lisp (+ hi 1))))
          (then (reply-to customer
                  (reply-keyword (in-lisp (* lo hi)))))
          (else
           (let ((average (in-lisp (floor (in-lisp (+ lo hi)) 2)))
                 (cont
                  (let ((unique-value (create null-script)))          ;create unique value
                    (create cont1 customer reply-keyword              ;create continuation with slots
                            unique-value unique-value unique-value))))) ;for returned values
             (request self (:do lo average) sponsor cont ':v1)
             (request self (:do (in-lisp (+ average 1)) hi)
               sponsor cont ':v2))))))))
    inline-rangeproduct-continuation-behavior-1))

(defequate inline-rangeproduct-continuation-behavior-1
  (script
   ((customer reply-keyword v1 v2 unique-value) ()
    (is-reply (:v1 v1)
      (if (== v2 unique-value)
        (then (update self (3 v1)))
        (else (reply-to customer
                (reply-keyword (in-lisp (* v1 v2)))))))
    (is-reply (:v2 v2)
      (if (== v1 unique-value)
        (then (update self (4 v2)))
        (else (reply-to customer
                (reply-keyword (in-lisp (* v1 v2)))))))))
```

Figure 6-12: Example of concurrent transactions and a joining customer.

start processing using the values; for example, the rangeproduct continuation multiplies the two numbers returned (v1 and v2) and replies with the product to the customer of the request to rangeproduct.

One possible alternative mechanism for keeping track of when all replies have been received is to keep a count of the replies received. However, this mechanism isn't as secure as the above mechanism; if, by some programmer's error, two replies are received for the same handler, then the joining customer may proceed even though some values haven't yet been received. Another alternative is to keep track of which replies have been received through some sort of bit vector; if there is a fast way of setting bits and testing the whole bit vector, then it may be more time efficient. However, if you use a integer as a bit vector, the compiler must be careful not to overflow the number of bits in the integer when there are many acquaintances.

The synchronization operation is one place where additional constructs may improve the efficiency of an implementation. For example, a unique "unbound" value which cannot be returned in messages could be provided, so that the overhead of creating a unique actor each time need not be incurred.

6.4.1 Identifying Joining Replies: Reply Keywords

Note the use of reply keywords in the inline-rangeproduct example. For each of the concurrent transactions, a different reply keyword is used (e.g. :v1, :v2). The joining customer distinguishes each reply by its reply keyword; each transaction has a separate reply handler which processes the reply from that transaction.

Since both the customer and the reply keyword identify what should happen to the reply, both must be supplied with every request. The extra reference to a reply keyword is additional communication overhead in every request message and additional storage overhead for every continuation which must remember the reply keyword with the customer, so the question arises whether there is another method of making sure results from concurrent transactions may be distinguished which avoids this overhead. In particular, the reply keywords are unnecessary for transactions which don't return to a joining continuation, so perhaps requiring them for all transactions may degrade the performance of the system.

The rationale for deciding to require reply keywords for all transactions is twofold: first, concurrent transactions are very common, and second, the other known method for identifying transactions incurs even more overhead. Joining continuations occur whenever an Acore expression has multiple ask expression parameters, or where a *let* has multiple arms containing ask expressions.

The other known method for identifying transactions is to use "tagging customers". Basically, the idea is to give each transaction a tagging customer which receives the final reply, and forwards the reply to the joining customer with the correct tag (figure 6-13).

Which of these solutions is optimal will depend upon the characteristics of the implementation and the characteristics of the application. The reply keyword method requires a little extra overhead for

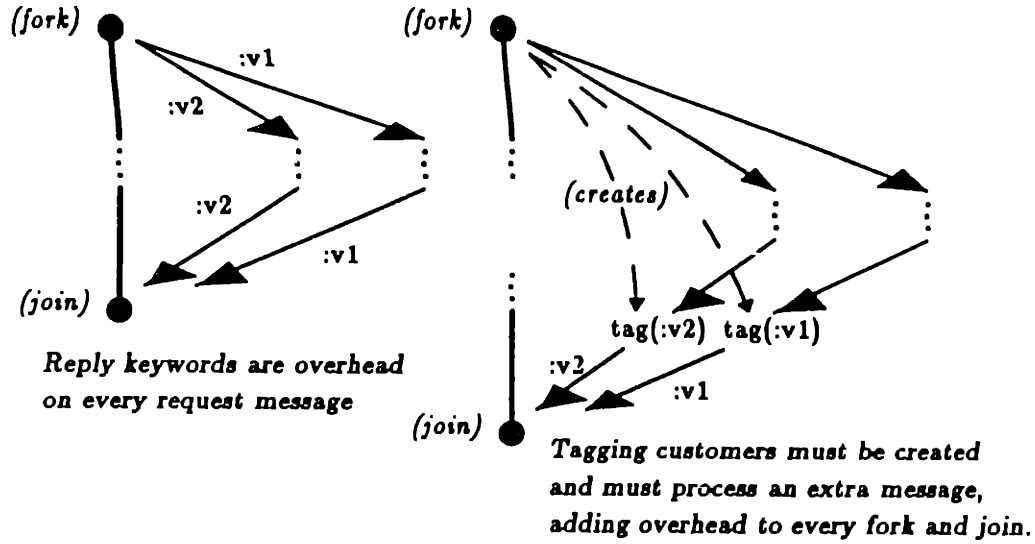


Figure 6-13: Tagging customers

every transaction. The tagging method requires an allocation/deallocation overhead for the tagging customers, but only when they are needed. It also requires the overhead of handling and initiating an extra message. In our current emulator, allocation/deallocation and the handling and initiating of an extra message are relatively expensive compared to the overhead of increasing the size of requests slightly, and since a large fraction of the customers in programs are joining customers, the reply keyword method has turned out to be more efficient. As we noted earlier (figure 6-9), the reply keywords also provide some error detection against a duplicate reply masquerading as the result of a subsequent transaction when the continuation is reused. Therefore we have chosen to use the reply keyword method in our current system.

6.4.2 Concurrent Sequences of Ask Expressions

In the inline-rangeproduct example, the two concurrent expressions were simple ask expressions which required only a continuation to receive the value. However, this simple solution doesn't cover the complications of all possible concurrent transactions, so we will now look at the case where the concurrent expressions may require separate continuations to perform intermediate processing before the final values are returned.

Consider the following program fragment from a quicksort program, abstracted into a function:

```
(lambda (left-lessers left-greaters
         right-lessers right-greaters)
 (:append
  (quicksort
   (:append left-lessers right-lessers))
  (quicksort
   (:append left-greaters right-greaters))))
```

The two calls to quicksort are concurrent, but each of them requires that an :append be made first. In each case, a continuation is required to receive the result from the append and make the call to quicksort.

It may be possible for a continuation to do double duty as both the joining continuation and as one of the concurrent transactions continuations. However, since the transactions are concurrent, the double duty continuation would have to be prepared to receive the replies from the other transactions at any time. To simplify the behaviors, I have chosen instead to create continuations for each of the branches; at the end of each branch, the last transaction replies to the joining continuation just as in the simple case (figure 6-14).

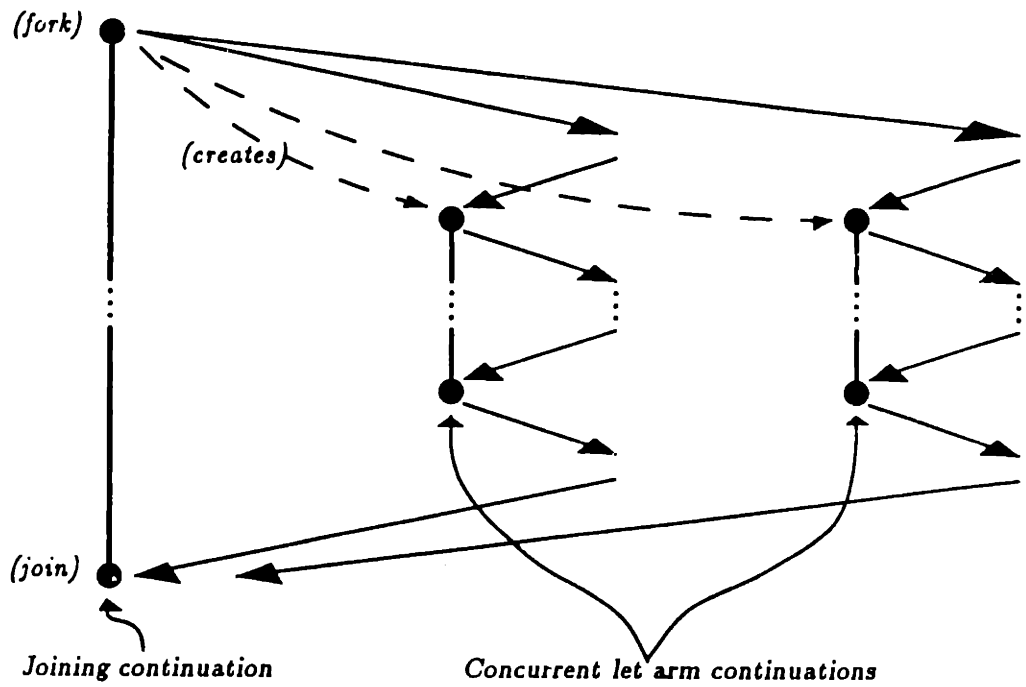


Figure 6-14: Pictorial depiction of joining and concurrent continuations.

It may also be possible for a continuation to do multiple duty, serving as the continuation for several concurrent expressions at once, e.g. serving both of the calls to :append and quicksort. However, this approach also results in complex behaviors, especially when the concurrent expressions are complicated, since the common continuation must keep track of the progress of several independent transactions. Therefore the most general approach is to give each concurrent transaction its own con-

tinuation; separating the continuations also permits the continuation actors to be distributed on separate processors.

```
(defequate quicksort-fragment-behavior
 (script
  (() (null-script jcont-script cont1 cont2)
   (is-request ((:do left-lessers left-greaters
                     right-lessers right-greaters) :unserialized)
    (let ((jcont (let ((unique-value (create null-script)))
                   (create jcont-script reply-keyword customer
                       sponsor unique-value unique-value
                       unique-value))))
     (request left-lessers (:append right-lessers)
       sponsor (create cont1 jcont sponsor) :v1)
     (request left-greaters (:append right-greaters)
       sponsor (create cont2 jcont sponsor) :v2))))
  null-script quicksort-fragment-joining-continuation-behavior
  quicksort-fragment-continuation-behavior-1
  quicksort-fragment-continuation-behavior-2))
                                                  ; First concurrent branch continuation
(defequate quicksort-fragment-continuation-behavior-1
 (script
  ((jcont sponsor) (quicksort)
   (is-reply (:v1 v1)
    (request quicksort (:do v1) sponsor jcont :v3)))
  quicksort))
                                                  ; Second concurrent branch continuation
(defequate quicksort-fragment-continuation-behavior-2
 (script
  ((jcont sponsor) (quicksort)
   (is-reply (:v2 v2)
    (request quicksort (:do v2) sponsor jcont :v4)))
  quicksort))
                                                  ; Joining continuation
(defequate quicksort-fragment-joining-continuation-behavior
 (script
  ((reply-keyword customer sponsor v3 v4 unique-value) ()
   (is-reply (:v3 v3)
    (if (== v4 unique-value)
     (then (update self (4 v3)))
     (else (request v3 (:append v4) sponsor customer reply-keyword))))
   (is-reply (:v4 v4)
    (if (== v3 unique-value)
     (then (update self (5 v4)))
     (else (request v3 (:append v4) sponsor customer reply-keyword)))))))
```

Figure 6-15: A quicksort fragment in Pract illustrating concurrent continuations.

Thus, the code fragment above may be implemented using the Pract behaviors of figure 6-15.

Note that the concurrent continuations (cont1 and cont2) are only involved with the intermediate stages of the concurrent expressions. The first request is made by the leading actor which makes the fork, and the final reply is returned directly to the joining continuation. This technique is also depicted in figure 6-14. Also note that the reply keyword for the final reply to the joining continuation is compiled into the continuation's behavior, but the joining continuation itself is created at run time and therefore must be remembered as an acquaintance to each of the concurrent continuations.

We have now covered the basic techniques needed for expressing ask expressions in Pract. The basic points can be summarized as follows: when a *leading actor* receives a request, it may create *continuation actors* to wait for the replies from intermediate transactions and continue processing once the replies are received. Sequential ask expressions may save allocation overhead by reusing the same continuation, changing its behavior for each transaction. Concurrent ask expressions require a *joining continuation* which resynchronizes transactions by waiting until all replies have been received before continuing. Concurrent continuations for each branch of the fork may be required if the expressions involve more than one ask expression. Finally, the joining continuation must be able to identify the transactions; we have chosen to supply reply keywords with each request because they are the most efficient and elegant solution we have found in our current emulator.

6.5 Serialized Handlers

Another common communication pattern occurs when an actor needs to compute values before it can specify a replacement behavior to process its next message. For example, a bank account actor may need to compute its new balance before it can specify its new behavior. To accomplish this, the leading actor takes on the *insensitive behavior*.

The insensitive behavior simply queues the messages it receives. Once the replacement behavior has been computed in a continuation, then a message is sent to the leading actor with the insensitive behavior, specifying what behavior to take on next. The leading actor then takes on the new behavior and releases the queued messages, ready to accept the next message. This process — of accepting a message, taking on the insensitive behavior, waiting for a replacement behavior, and then taking on the new replacement behavior to process the next message — effectively *serializes* the messages accepted into a total ordering based on the arrival order.

Since this pattern of behavior is common, it is hidden under the Pract implementation where it can be optimized. When a Pract actor receives a message and processes it with a handler which has not been declared *unserialized*, it automatically takes on the insensitive behavior. To release the actor from its insensitive state, a *replace* or *update* message must be sent to it. A *replace* message specifies the complete replacement behavior the insensitive actor should take on next, including both the script and the acquaintances. *Update* is an optimization for the common case where only some of the acquaintances or script are to be changed; the unchanged acquaintances and script remain as they were before the actor

took on the insensitive behavior. In previous examples *update* has been frequently used by continuations to specify their own replacement behaviors.

```
(defequate account-behavior
 (script
  ((balance) (deposit-cont withdrawal-cont)
   (is-request ((:balance) :unserialized)
    (reply-to customer (reply-keyword balance)))
   (is-request (:deposit amount)
    (let ((cont (create deposit-cont self customer reply-keyword)))
     (request balance (:+ amount) sponsor cont :new-balance)))
   (is-request (:withdrawal amount)
    (let ((cont (create withdrawal-cont self amount
                        sponsor customer reply-keyword)))
     (request balance (:- amount) sponsor cont :new-balance))))
  account-deposit-cont-behavior-1 account-withdrawal-cont-behavior-1))

(defequate account-deposit-cont-behavior-1
 (script
  ((leading-actor customer reply-keyword) ()
   (is-reply (:new-balance new-balance)
    (update leading-actor (1 balance))
    (reply-to customer (reply-keyword leading-actor))))))

(defequate account-withdrawal-cont-behavior-1
 (script
  ((leading-actor amount sponsor customer reply-keyword) (cont3)
   (is-reply (:new-balance new-balance)
    (update self (0 cont3) (3 new-balance))
    (request new-balance (:>= 0) sponsor self :v1)))
  account-withdrawal-cont-behavior-2))

(defequate account-withdrawal-cont-behavior-2
 (script
  ((leading-actor amount new-balance customer reply-keyword) ()
   (is-reply (:v1 v1)
    (if v1
     (then (update leading-actor (1 new-balance))
           (reply-to customer (reply-keyword amount)))
     (else (update leading-actor)
           (complain-to customer (:overdraft reply-keyword))))))))
```

Figure 6-16: A Bank Account behavior in Pract.

For example, consider the simple bank account behavior of figure 2-1 (page 14) translated into Pract in figure 6-16. The :balance handler is unserialized, so the account remains unlocked after the message is processed. The :deposit and :withdrawal handlers are serialized, so the account actor (the leading actor) becomes insensitive until the continuation sends an update message to it. Here we find

another reason why *update* messages refer to acquaintances by position. Since the *update* may be sent from a continuation script which is completely separate from the leading actor's script, acquaintance names may be meaningless.

From this example, we can see that Acore commands are easily translated into Pract. When all the parameters to the command have been evaluated, the continuation may simply issue the equivalent Pract command. Since *update* is the equivalent of an Acore *ready*, a *ready* command may be translated into an *update* and processed like any other command.

All of the continuations so far only have reply handlers; none handle exceptional responses. We turn now to see how continuations need to be augmented to handle complaints.

6.6 Complaints and Complaint Handling

Complaints are exceptional responses, sent instead of reply messages in exceptional situations. In order to handle complaints, a continuation must have complaint handlers in addition to its reply handlers. The primary issue concerning complaints is figuring out what complaint handlers a continuation should have, but there are also some subtle complications for concurrent transactions.

Complaint handling is specified in Acore through the *let-except* statement; the complaint handlers specify what to do with a complaint generated in one of the arms of the *let-except*. If the expression in any of those arms cannot be evaluated because a complaint was returned rather than a value, the appropriate exception handler should be invoked — therefore, every continuation receiving a value from those expressions must have the complaint handlers. Of course, if the complaint handlers have continuation behaviors, they may be shared by the continuations so there isn't too much duplication of code. Within a script, *let-except* statements may be nested; in this case, a continuation should have all the complaint handlers which apply to its context and which aren't shadowed by a closer handler for the same complaint. These points are easier to see from examples, so let's look at a few.

All continuations should have some default complaint handling which at the least forwards the complaint to the customer; otherwise if an unexpected error occurs (aren't most errors unexpected?) the transaction will die mysteriously rather than being aborted cleanly. The normal expression context handler provides this, so actually the Acore script

```
(script ()
 ((:do (parameters...))
  expression body...))
```

is equivalent to the following Acore script:

```
(script ()
 (is-request (:do (parameters...))
  (let-except ((value (let () expression body...)))
              (except-when
               ((some-error (&rest parameters))
                (complain-to* customer
                  (some-error reply-keyword parameters))))
    (reply-to customer (reply-keyword value)))))
```

Let-except handlers look like message handlers because the pattern forms the pattern for the complaint handler in continuations. For example, in figure 6-17, the :overflow exception handler in the Acore code is translated into the :overflow complaint handler in the following Pract behaviors. We can check the example against three of the points we mentioned above.

- The exception handling applies if the expression in an arm cannot be evaluated because of a complaint, so the complaint handlers are present in both continuations.
- The continuation behavior for the exception handler is shared by the continuations which use it.
- The continuations receive values for expressions which are in the context of both the explicit handler for :overflow and the implicit handler for some-error which covers the body of the lambda. Since the inner handler doesn't override the outer one, both are present in the continuations.

Two other interesting things to note: Even the transaction within the inner :overflow exception handler has complaint handling from the outer some-error exception handler present in the continuation. Also, the tail recursion optimization has been applied to this continuation; it can be applied as long as the both the reply handlers and the complaint handlers merely forward the message received to the customer with the reply keyword.

Complaint handling is more complicated when concurrent transactions are involved. In a concurrent *let-except*, a complaint in any of the arms prevents it from proceeding to the body; instead, the exception handler is invoked and any other values (or complaints) returned from the concurrent transactions are ignored. Because of this synchronization performed by the joining continuation, when there are continuations for the arms of the *let-except*, these concurrent continuations must forward the complaint to the joining continuation (unless they have a more local complaint handler). Again, an example will help make things clear.

In figure 6-18, the explicit :overflow exception handler of the Acore code applies to the both the concurrent expressions and their subexpressions. This expression is implemented in Pract in figure 6-19. Since the concurrent expressions require continuations, the arm continuations as well as the the joining continuation need to handle the possibility of a complaint. However, the joining continuation must perform some synchronization so that no more than one response is returned. If the arm continuations were to process the complaints, then if several arms complained, several complaints might be returned as the value of the *let-except*. Therefore, the arm continuations don't directly process the complaints they

The exception handler in this Acore code:

```
(lambda (i r v0)
 (let-except ((v (:+ v0 (:* i r))))
             (except-when
              ((:overflow ())
               (:+ v0 (:* imax r))))
  v))
```

Produces the complaint handlers in the continuations of this Pract code:

```
(create (script (() (c1)
                    (is-request ((:do i r) :unserialized)
                     (request i (:* r) sponsor
                      (create c1 customer reply-keyword
                       sponsor v0) :v)))
                   continuation-1))
(defequate continuation-1
 (script
  ((customer reply-keyword r sponsor v0) (c2 imax ec1)
   (is-reply (:v1 v1)
    (update self (0 c2))
    (request v0 (:+ v1) sponsor self :v))
   (is-complaint (:overflow ignore)
    (request imax (:* r)
     sponsor (create ec1 customer reply-keyword sponsor v0) :v2))
   (is-complaint (some-error &rest parameters)
    (complain-to* customer
     (some-error reply-keyword parameters))))
  continuation-2 exception-continuation-1 imax))
(defequate continuation-2
 (script
  ((customer reply-keyword r sponsor v0) (imax ec1)
   (is-reply (:v v)
    (reply-to customer (reply-keyword v)))
   (is-complaint (:overflow ignore)
    (request imax (:* r)
     sponsor (create c3 customer reply-keyword sponsor v0) :v2)))
   (is-complaint (some-error &rest parameters)
    (complain-to* customer
     (some-error reply-keyword parameters))))
  imax exception-continuation-1))
(defequate exception-continuation-1
 (script
  ((customer reply-keyword sponsor v0) ()
   (is-reply (:v2 v2)
    (request v0 (:+ v2) sponsor customer reply-keyword))
   (is-complaint (some-error &rest parameters)
    (complain-to* customer
     (some-error reply-keyword parameters))))))
```

Figure 6-17: Let-Except specifies complaint handling.

```
(lambda (l theta)
 (let-except ((h (:* l (tan theta)))
              (d (:/ l (cos theta))))
             (except-when
              ((:overflow ())
               0))
  (:+ h d)))
```

Figure 6-18: *Let-except* with concurrent arms.

receive, but instead forward them to the joining continuation by making a request to the complaint-identity actor. The complaint-identity returns a complaint identical to the request message that it receives; the primary reason for this indirection is to simplify the transaction structure for debugging.

Once the joining continuation (jcont-script-1) receives the complaint, it first checks to see if it has already received a previous complaint. If so, then it just ignores the complaint. If not, then it updates itself with :abort to remember that it has received a complaint, and may process the complaint, either returning 0 for :overflow complaints or just forwarding other complaints.

The perceptive reader may notice that the reply keyword is always returned as the first parameter to the complaint. This allows future joining continuations to distinguish complaints from different transactions and process them differently; this feature is currently unused.

Now that we have seen how complaint handling is incorporated into single, joining, and arm continuations, we can put aside this complication again. We turn now to *script* expressions to see how the lexical scoping and referential transparency of Acore maps into Pract.

6.7 Scripts

There are three relatively simple issues to consider concerning scripts: how we combine a script and its continuations into one expression, how a guardian for a script is implemented, and how a script captures the values of its free identifiers. We examine each of these in turn below.

In the examples so far I have presented Pract behaviors as a series of *defequate*'s, since this is the clearest to read. However, Acore *script* expressions appear nested inside of other scripts, so the Pract equivalent of an Acore *script* expression must be an expression. The simple answer to this is to define the continuations as a series of nested *let*s, making sure that each continuation is defined before it is used.

For example, the script from figure 6-16 contained several continuations, which can be nested in *let*s as shown below.

```
(create (script
      (() (c1 c2 c3 cos tan null-script)
       (is-request ((:do l theta) :unserialized)
        (let ((jcont (let ((unique-value (create null-script)))
                       (create c3 reply-keyword customer
                           sponsor unique-value
                           unique-value unique-value))))
          (request tan (:do theta) sponsor
            (create c1 jcont sponsor l) :v1)
          (request cos (:do theta) sponsor
            (create c2 jcont sponsor l) :v2))))
      arm-cont-script-1 arm-cont-script-2 jcont-script-1
      cos tan null-script))
(defequate arm-cont-script-1
  (script ((jcont sponsor l) (complaint-identity)
       (is-reply (:v1 v1)
         (request l (:* v1) sponsor jcont :h))
       (is-complaint (keyword ignore &rest parameters)
         (request* complaint-identity (keyword parameters)
           sponsor jcont :h)))
   complaint-identity))
(defequate arm-cont-script-1
  (script ((jcont sponsor l) (complaint-identity)
       (is-reply (:v2 v2)
         (request l (:/ v2) sponsor jcont :d))
       (is-complaint (keyword ignore &rest parameters)
         (request* complaint-identity (keyword parameters)
           sponsor jcont :d)))
   complaint-identity))
(defequate jcont-script-1
  (script
   ((reply-keyword customer sponsor d h unique-value) ()
    (is-reply (:d d)
      (if (or (== unique-value ':abort) (== h unique-value))
        (then (update self (4 d)))
        (else (request h (:+ d) sponsor customer reply-keyword))))
    (is-reply (:h h)
      (if (or (== unique-value ':abort) (== d unique-value))
        (then (update self (5 h)))
        (else (request h (:+ d) sponsor customer reply-keyword))))
    (is-complaint (:overflow ignore)
      (if (== unique-value ':abort)
        (then (update self))
        (else (update self (6 ':abort))
          (reply-to customer (reply-keyword 0)))))
    (is-complaint (some-error ignore &rest parameters)
      (if (== unique-value ':abort)
        (then (update self))
        (else (update self (6 ':abort))
          (complain-to* customer
            (some-error reply-keyword parameters)))))))))
```

Figure 6-19: Example of figure 6-18 implemented in Pract.

```
(defequate account-behavior
 (let ((account-withdrawal-cont-behavior-2 (script ...)))
  (let ((account-withdrawal-cont-behavior-1
         (script ... account-withdrawal-cont-behavior-2)))
   (let ((account-deposit-cont-behavior-1 (script ...)))
    (script ... account-deposit-cont-behavior-1
           account-withdrawal-cont-behavior-1)))))
```

Since no loops may be expressed within an Acore behavior (iteration is expressed with tail recursion), the dependencies between continuation behaviors must form a directed acyclic graph. Since the graph is acyclic, there is always a linear ordering in which dependencies are satisfied, so this nesting is always possible.

The guardian is just an actor which accepts :create and :replace messages concerning the Pract script:

```
(defequate account-guardian
 (let ((account-behavior (let (...) ...)))
  (create
   (script
    (() (account-behavior)
     (is-request ((:create initial-balance) :unserialized)
      (let ((account (create account-behavior initial-balance)))
       (reply-to customer (reply-keyword account))))
     (is-request ((:replace actor initial-balance) :unserialized)
      (replace actor account-behavior initial-balance)
      (reply-to customer (reply-keyword actor))))
    account-behavior))))
```

The guardian encapsulates the raw script, controlling all create's and replace's using the raw script. No direct references to the script are available outside the guardian, so there can be no forwarding actor problems with use of the script. The guardian's handlers require a specific number of parameters which correspond to the acquaintances of the script, so it is impossible to create an actor with the wrong number of acquaintances.

There are a few cases where the guardian may be safely optimized away. For example, if the *script* is used directly in a *create* expression, then as long as the *create* expression has the right number of parameters, it is safe to omit the guardian and create the actor directly, saving a transaction and the allocation of the guardian. Thus,

```
(:create <expression creating guardian and script>
         <correct number of acquaintances...>)
```

may be optimized as:

```
(create <expression creating raw script>
        <correct number of acquaintances...>)
```

The third question about scripts concerned how the lexical scoping of Acore scripts is implemented in Pract. This also has a simple answer: since identifiers are referentially transparent, scripts are simply created with copies of the bindings of the free identifiers as part of their local state. When scripts are

nested, any free identifiers of the inner scripts must be found in the enclosing scripts, either as local identifiers or free identifiers. Free identifiers are declared in Pract scripts, so this propagation of free identifiers becomes explicit when the nested scripts are translated into Pract. For example, the expression

```
(lambda (x)
 (lambda (y)
  (lambda (z)
   (f x y z))))
```

which is expressed in Acore without macros as

```
(:create (script ()
          ((:do (x) :unserialized)
           (:create (script ()
                     ((:do (y) :unserialized)
                      (:create (script ()
                                ((:do (z) :unserialized)
                                 (f x y z)))))))))))
```

may be expressed in Pract (optimizing away guardians) as follows:

```
(create
 (script
  (() (f)
   (is-request ((:do x) :unserialized)
    (reply-to customer
     (reply-keyword
      (create
       (script
        (() (f x)
         (is-request ((:do y) :unserialized)
          (reply-to customer
           (reply-keyword
            (create
             (script
              (() (f x y)
               (is-request ((:do z) :unserialized)
                (request f (:do x y z)
                 sponsor customer reply-keyword)))
              f x y))))))
         f x))))))
  f))
```

Each script is created with the values of its free identifiers; unlike other lexically scoped languages, since identifiers are referentially transparent (don't change value) there is no need to refer to the values of identifiers indirectly through an environment. Closures are therefore inexpensive to create.

6.8 Special Forms

The rest of this chapter is about the implementation of the special forms in Acore. We will look at *future*, *delay*, *race*, *values*, and *with-sponsor*.

6.8.1 Futures

A future is an actor which behaves as an insensitive actor, queuing all its messages, until a value is known. Then it becomes a forwarding actor to the value. There are two simple issues concerning futures: how to implement the futures themselves, and deciding the behavior of a future whose expression produces a complaint.

One possible implementation of futures is to take advantage of the insensitive behavior provided by Pract. Using this implementation, the future expression is almost a macro which expands in Acore as follows:

```
(future expression) →

(let ((v6 (create-locked future-script)))
  (let-except ((value expression))
          (except-when
           ((some-error (&rest parameters))
            (replace v6 complainer-script
                     some-error parameters)))
    (replace v6 forwarder-script value))
  v6)
```

Parallelism arises from performing the *let-except* concurrently with returning v6. In this implementation, the behavior represented by future-script doesn't really matter since the future is created in the insensitive state and queues all messages until either a forwarder or complainer replacement behavior is specified. However, it is useful to give the future a script which identifies it as a future for debugging purposes.

When the *expression* successfully computes a value, the future becomes a forwarding actor which forwards all messages to the value. However, if *expression* should return an exceptional response (a complaint), what is the suitable response for the future? There are several possibilities: it could simply remain insensitive; it could perform the exception processing of the surrounding context, hoping that would produce a value; or it could become a complainer which responds to every message it receives with a duplicate of the complaint.

We have chosen the last alternative. If we were to take the first alternative, the future becomes a mysterious black hole, queueing all messages without releasing any clues about what went wrong. The second alternative may invoke invalid exception processing: Exception handling traps the cases where an expression produced a complaint instead of a value, but the future expression has already produced a value — the future — and that value may have already been returned out of that context. Thus, if the exception handler is designed to *ready* some actor if the transaction fails to produce a value, invoking it when the future receives a complaint will produce an error by trying to *ready* an actor which isn't insensitive. Thus, the third alternative seems the most reasonable; it is noisy enough to help locate errors, and it doesn't produce any more errors by itself. The complainer script mentioned in the above code gives the future the behavior of complaining for every message it receives.

A future expression without a value simply produces the insensitive actor: in this case it is the program's responsibility to specify the replacement behavior, as in the recursive let example of figure 4-13 (page 41).

6.8.2 Delay Futures

A delay encapsulates a closure for an expression, waiting to evaluate it until it is needed. When the delay receives a message, it evaluates the closure, and when the value is received, it forwards the initial message to the value and becomes a forwarding actor to the value. Delay isn't difficult to implement in Acore, since closures are easily created. The behavior of a delay when the expression evaluates to a complaint is the same as for a future, for the same reasons. The only subtle issue concerning delay is what sponsor is used to evaluate the expression.

The delay behavior could be implemented something like the following Acore code:

```
(delay expression) →

(create
 (script ()
  (is-request (keyword &rest parameters)          ;When the first message is received
   (let-except ((value expression))                                ;First compute value
          (except-when
           ((some-error (&rest parameters))
            (:replace complainer-script self some-error parameters)
            (complain-to* customer
              (some-error reply-keyword parameters))))
    (:replace forwarder-script self value)              ; then become a forwarding actor
    (request* value (keyword parameters)              ; and forward first message to value
      sponsor customer reply-keyword)))))
```

Alternatively, to make the closure even more explicit, the delay behavior could be implemented as the following:

```
(delay expression) →

(create delay-script (lambda () expression))
```

where delay-script describes the following Acore behavior:

```
(defequate delay-script
 (script (closure)
  (is-request (keyword &rest parameters)          ;When the first message is received
   (let-except ((value (closure)))                                 ;Evaluate closure
          (except-when
           ((some-error (&rest parameters))
            (:replace complainer-script self some-error parameters)
            (complain-to* customer
              (some-error reply-keyword parameters))))
    (:replace forwarder-script self value)              ; then become a forwarding actor
    (request* value (keyword parameters)              ; and forward first message to value
      sponsor customer reply-keyword)))))
```

Note that in either case the *delay* expression differs from the *future* expression in that in the case of a *future*, the *expression* is evaluated under the sponsorship found in the context where the future expression is located, whereas in the case of a *delay* the *expression* is evaluated under the sponsorship of the sponsor of the first incoming request. The reason is that by the time the *delay* gets its first message, the sponsor of the context where it was created may have run out of funds. Also, if the *future* like sponsorship arrangement is desired, it is easy to specify using *with-sponsor*:

```
(delay (with-sponsor sponsor
         expression))
```

6.8.3 Race

A *race* expression evaluates its subexpressions concurrently, forming a list of the values returned in the order they are returned. This queue-like list is formed incrementally, so each value is available soon after it is returned. How can we implement this behavior in Pract?

Since the values are queued in the order they are returned, they must return to a single joining customer which puts them into the queue according to their arrival order. Unlike the joining customer of a parallel *let*, however, *race* does not need to distinguish the values returned; it only needs to count them so that the queue can be terminated when all values are returned. One way to implement the undetermined list is as a future; as values are received the list can be incrementally created. Thus this implementation looks something like figure 6-20.

When a value is returned to the *race* continuation, the future becomes a cons cell whose car is the value returned and whose cdr is the future for the rest of the list. If there are no more values, then the *race* continuation terminates the list with nil instead. Complaints are handled in a manner similar to how they are handled in futures and delays: a complainer actor takes the place of the value in the list.

6.8.4 Multiple Values

Values is a simple form for returning multiple values from an expression context. Multiple values may be bound with a let which supplies multiple identifiers; for example:

```
(let (((a b) (values b a)))
  ...)
```

is a silly way to exchange the values of a and b for the body of the let.

When the values are being returned from a handler, a multiple value reply can be used to implement the multiple values directly. For example, the returning the multiple values car and cdr at the end of this Acore script:

```
(defequate cons-script
 (script (car cdr)
  ...
  ((:decompose () :unserialized)
   (values car cdr))))
```

The Acore expression:

```
(race expr1 expr2 ... exprN)
```

may be implemented as:

```
(let ((queue (future)))
 (let ((race-cont (create race-script N queue)))
   {expr1 expr2 ... exprN})
   queue)
```

where N *is the number of subexpressions and {expr1 expr2 ... exprN} are evaluated with* race-cont *as their continuation.*

Race-script *may be implemented like the following (#I indicates inline operations):*

```
(defequate race-script
 (script (n-left next-future)
  (is-reply (:value value)
   (if #I(<= n-left 1)
    (then (replace next-future cons-script value nil))
    (else (let ((new-next (future)))
       (replace next-future cons-script value new-next)
       (ready (n-left #I(- n-left 1))
           (next-future new-next))))))
  (is-complaint (some-error ignore &rest args)
   (let ((value (create complainer-script some-error args)))
    (if #I(<= n-left 1)
     (then (replace next-future cons-script value nil))
     (else (let ((new-next (future)))
       (replace next-future cons-script value new-next)
       (ready (n-left #I(- n-left 1))
           (next-future new-next)))))))))
```

Figure 6-20: Implementation of *race*.

may be implemented as a multiple value reply at the end of this Pract script:

```
(defequate cons-script
 (script
  ((car cdr) ()
   ...
   (is-request (:decompose () :unserialized)
    (reply-to customer (reply-keyword car cdr))))))
```

However, when the values are being returned within a handler, as in the silly swapping example, the simplest way to implement *values* is as an ask expression to a reply-identity actor which returns the multiple values.

```
(reply-identity b a)
```

(*Complaint* can be implemented in a similar manner.) The Reply-identity actor can be implemented as follows:

```
(defname reply-identity
 (create
  (script
   (() ()
    (is-request (:do (&rest parameters) :unserialized)
      (reply-to* customer (reply-keyword parameters)))))))
```

Binding multiple values is easy as long as the values are being returned to a continuation; the reply handler can simply bind multiple parts of the message in the same way that request handlers bind multiple parameters. This is why the easiest way to implement multiple values within a script is to create an ask expression. So for example, our silly swapping example can be compiled as:

```
...
(request reply-identity (:do b a)
  sponsor (create silly-swap-cont-1 ...) :v1))...

(defequate silly-swap-cont-1
 (script
  ((...) (...)
   (is-reply (:v1 a b)
     <use swapped a and b> ...)
  ...)
 ...))
```

One final point about multiple values. In Lisp, multiple values may be returned even if the program only expects one value; only the first value is used and the rest are ignored. This effect can be created in Pract by allowing further values in the reply handler, but ignoring them. If this route is taken, all the reply handlers in our examples, which have looked something like:

```
(is-reply (:v2 v2)
  ...)
```

would instead look like

```
(is-reply (:v2 v2 &rest ignore)
  ...)
```

6.8.5 Sponsorship

In Acore, sponsorship of the transactions specified by the ask expressions comes implicitly from the context. By default the subtransactions initiated by a handler are sponsored by the sponsor of the incoming transaction which invoked the handler. Sponsorship may be specified by nesting the expressions to be specially sponsored in a *with-sponsor* form. In contrast, sponsorship in Pract is specified explicitly in each request command. It is the responsibility of the implementation which runs Pract programs to charge sponsor ticks, generating requests to the sponsor for more ticks when necessary; since ticks are charged for every message, this method permits the most flexibility for the implementation to gain efficiency. (See Appendix A for an explanation of how our emulator does it.) Thus, dealing with sponsorship in translating Acore to Pract is a matter of making sure that each request is supplied with the correct sponsor.

The *with-sponsor* form specifies a sponsor to be used within its body. Since sponsorship is a matter of context, it is possible to imagine an implementation of this form which operates similarly to a *let*; for example, the form:

(*with-sponsor sponsor-expression*
  body...)

would be translated into something like:

(*let* ((v8 *sponsor-expression*))
  <*body using* v8 *as the sponsor...*>)

6.9 Summary

In this chapter we have looked at the how Acore programs can be implemented in terms of primitive actors. Primitive actors, such as those written in Pract, are suitable for being run by an parallel architecture since they process messages using only their local state. Thus they do not block waiting for remote information and they may be easily migrated between processors for load balancing. The challenge, then, was to find ways of implementing the complex control expressed with ask expressions in Acore by organizing the behaviors of these simple actors.

- We found that the waiting of an ask expression can best be expressed by creating a *continuation* to accept the reply and continue the transaction. The continuation separates context information for recursive or concurrent transactions. Sequential ask expressions may share the same continuation if it changes behavior between each subtransaction. Concurrent ask expressions require a *joining continuation* which resynchronizes the concurrent transactions, waiting for all replies before continuing. Joining continuations are common, so *reply keywords* are used in all transactions so that joining continuations can distinguish replies. Sometimes each of the concurrent branches may require its own continuation.
- Acore permits replacement behaviors to be computed, while primitive actors require that a replacement behavior be specified for each event. We found that to implement Acore behavior replacement, the leading actor needs to take on the *insensitive behavior*, queueing messages until the new behavior is known. Since this behavior is common, Pract behaviors automatically specify the insensitive behavior, and *replace* or *update* messages are sent to the insensitive actor to specify its new behavior.
- Complaints are exceptional responses, so complaint handling is performed by the complaint handlers of customers. An exception handler applies to all the expressions in its scope, so the complaint handlers are present in all the continuation behaviors which implement the expressions. Multiple complaint handlers may be necessary when there are multiple exception handlers or the exception handling is nested. Concurrent transactions complicate this picture slightly; since the joining customer may only continue once, it aborts as soon as the first of the concurrent transactions complains. The arm continuations must forward complaints to the joining continuation to perform this synchronization.
- We found that because of referential transparency, the lexical scoping and closure of scripts in Acore is easy to implement in terms of primitive actors — the primitive behaviors may simply hold a copy of the reference the values of identifiers. Script expressions may be implemented by nesting the continuation behaviors in *let*'s in a linear ordering. We also saw how Script guardians may be implemented.
- The special forms *future*, *delay*, and *race*, can be implemented by suitable programming of primitive actors, and we saw an implementation of each of these, taking advantage of the

insensitive behaviors provided by Pract. Each of these special forms returns a value before its subexpressions have been evaluated; we found that creating a complainer in place of the value seems to be the most reasonable thing to do when an expression returns a complaint instead of a value.

- The message passing paradigm makes multiple values easy to implement; returning multiple values is simply another instance of sending a message with several parameters. Thus multiple valued expressions are implemented with multiple value replies, and bound with multiple valued reply handlers.

Now that we have a good idea of what compiled Acore programs should look like at the level of primitive actors, it is time we looked at how a compiler which performs the transformation can be implemented.

Chapter Seven

The Acore Compiler

The task of the Acore compiler is to transform an Acore program into an equivalent Pract program. This is a relatively complex process, so first we will make an overview of the compilation process; then we will go through an example to get a rough idea how the compilation process works; and finally, once we have a sketch in mind about how things fit together, we can look at each stage of the compilation process in detail. Readers should feel free to go into as little or as much depth as they find interesting, and then skip to the summary at the end of the chapter — the detailed pass contains discussion of some issues which may be only significant to someone actually trying to understand this implementation or implement a similar compiler, and may be safely skimmed or skipped entirely.

7.1 Compiler Overview

To connect the world of Acore programs with the level of actors which parallel architectures deal with, the Acore compiler translates Acore code into primitive actors in Pract. The Acore compiler is organized approximately in several stages as pictured in figure 7-1. It is currently written in Lisp, and since Acore uses a fully parenthesized syntax similar to Lisp, it accepts Acore source code in the form of Lisp list structure returned by a Lisp reader, and emits Pract source code as Lisp list structure as well.

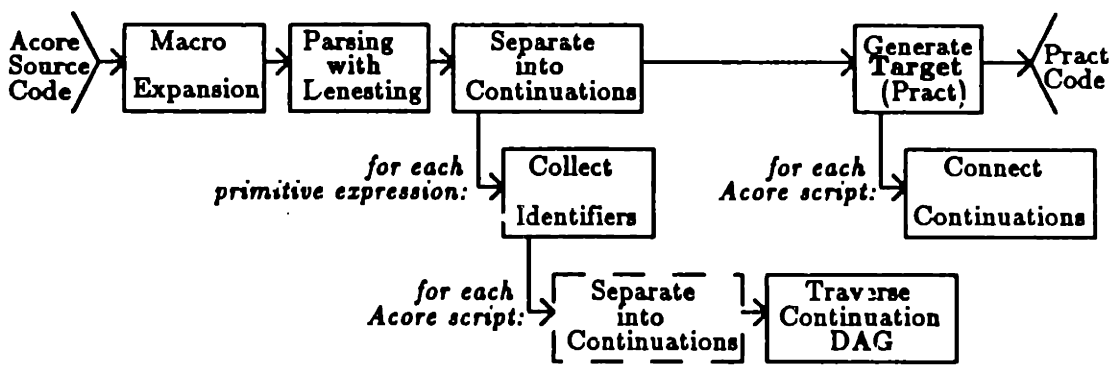


Figure 7-1: Compiler organization.

1. The *macroexpansion* phase performs source to source transformations on the source program according to the syntactic abstractions specified by the macro environment. Source code is manipulated in the form of list structure; the result is raw Acore code without any syntactic sugar.
2. The *parsing* phase of the compiler accomplishes several tasks: It checks the syntax of the forms being parsed; it uniquely identifies and classifies identifiers according to where they

are bound; it marks *continued* expressions and commands which contain ask expressions; it *denests* these continued forms to produce equivalent *let* forms; and it stores the information gained from the parse into a parse tree of *pnodes*.

3. The intermediate *icode* phase takes care of separating expressions into primitive actor behaviors connected by commands. It also collects the sets of identifiers referenced by the commands and expressions in each primitive behavior. During this process, it may recursively invoke the icode phase on script expressions.
4. For each script, the *traverse DAG* phase performs a depth first walk through the directed acyclic graph (DAG) of primitive behaviors, propagating local identifiers so that each identifier will be passed from continuation to continuation from the point where it was bound to the point where it is referenced. To do this, it must find which referenced identifiers are bound in a continuation and which should be passed from previous continuations. It also orders the continuation scripts so each will be defined before it is referenced.
5. Finally, the *Pract generation* phase walks the resulting graph, constructing the equivalent Pract commands and scripts to implement the behavior. For each Acore script, now transformed into nodes representing the leading and continuation behaviors, the *behavior connecting* phase first does a depth first non-redundant walk through the directed acyclic graph of continuations, ordering the acquaintances in the acquaintance lists and inserting the commands by which the continuations create or become the following continuations. Once the continuation nodes are completed, Pract code can be generated from them as well.

This overview may not be entirely clear initially, so it may help to reread it as the processing at each of the stages becomes clearer. To sketch in a rough idea of what's going on, we will now go through an example.

7.2 Example Compilation

For our example, we will compile the function rangeproduct. The source code for rangeproduct written in Acore is shown in figure 7-2. Two macros have been used which make this example easy to read: DefFunction abstracts definitions of function actors, and sequential cond* abstracts the syntax of a series of nested *if*'s. The function floor performs integer division.

7.2.1 Expanding Macros

The first stage of compilation is to expand the macros in this source code. The result of the macro expansion is shown in figure 7-3. The deffunction macro expands into a form which creates an actor which behaves as the function described, accepting :do messages with the parameters of the function and performing the body of the function to compute the reply. The resulting *script* expression with an expression context request handler expands into an *is-request* with *let-except* exception handling. We can also see the cond macro abstracts a series of nested *if*'s nicely.

7.2.2 Parsing and Denesting

The next phase of the compiler is the parse phase, which produces the parsed structures used by later stages of the compiler, and parses identifiers uniquely using an environment. While it builds the

Source:

```
(DefFunction RangeProduct (lo hi)
 (cond* ((:= lo hi)
         lo)
        ((:= lo (:- hi 1))
         (:* lo hi))
        (else (let ((average (floor (:+ lo hi) 2)))
                (:* (rangeproduct lo average)
                    (rangeproduct (:+ average 1) hi))))))
```

Figure 7-2: Source code for Rangeproduct

After Macro Expansion:

```
(defname rangeproduct
 (create
  (script ()
   (is-request
    (:do (lo hi) :unserialized)
    (let-except
       ((v0
         (if (:= lo hi)
          (then lo)
          (else (if (:= lo (:- hi 1))
                  (then (:* lo hi))
                  (else
                    (let ((average (floor (:+ lo hi) 2)))
                      (:* (rangeproduct lo average)
                          (rangeproduct (:+ average 1) hi)))))))))
       (except-when
        ((some-error (&rest args))
         (complain-to* customer
           (some-error reply-keyword args))))
      (reply-to customer
        (reply-keyword v0)))))))
```

Figure 7-3: RangeProduct after macro expansion

structure, it discovers which expressions are *continued*, i.e. which expressions either are ask expressions or contain ask expressions. An important part of this phase is *denesting* continued subexpressions.

Denesting takes continued subexpressions nested inside of other expressions and commands and pulls them out, forming an equivalent *let*. For example, denesting the expression:

```
(:+ (:* a a) (:* b b))
```

produces the equivalent expression:

```
(let ((v1 (:* a a))
      (v2 (:* b b)))
  (:+ v1 v2))
```

Note that a and b are not continued expressions and therefore are not denested from the multiplication expressions.

Most of the parsed structures have the same structural relations as the commands, expressions, and identifiers of the source code, only more information has been added. While it would be difficult to show the parsed structure which results from parsing rangeproduct, we can still see the basic effect of denesting by showing the result of denesting the source code directly. Figure 7-4 shows the how the rangeproduct example would look if denesting were performed directly on the source code.

7.2.3 Separating into Primitive Behaviors

The next stage is to separate this behavior into primitive actor behaviors, a leading actor behavior followed by continuation behaviors. The denesting performed by the parse phase transformed the code into a form suitable for this operation: the denested form makes it possible to traverse the parse structure encountering commands in the order they must be performed; each ask expression is separated into a *let* binding, making them easy to identify; and therefore the body of each continuation is separated into the body of a *let*. The result of separating the denested expressions of figure 7-4 is a set of primitive behaviors connected as shown in figure 7-5.

This graph shows the relations between the primitive behaviors produced to implement the rangeproduct function. The nodes are the individual primitive behaviors; the text to the right of each node describes the behavior of that node. Nodes are connected by lines representing the continuation relation, usually involving an ask expression; each line is labeled by the identifier whose value is being computed for the continuation. The dotted line indicates a relation where an ask expression is not needed to compute the value; instead, the body of the continuation (reply v0) is "pulled" into continuation where it is used directly on the value (reply lo). The forks with arcs indicate concurrent transactions due to a parallel *let*; forks without arcs show alternate paths due to decisions made by an *if*.

To transform the pnode structure from a form like that shown in figure 7-4 to this behavior graph, the *icode* phase performs the following steps. Let's mimic this separation phase to see what it does.

1. We create inode structures to represent the leading script and its handlers. For example, as we walk down through the code of figure 7-4, we create inode structures for the leading *script* and its *is-request* handler, represented by the top node of the graph in figure 7-5.
2. We walk the parse structure, keeping track of the current parsed expression or command, the current inode body in which to insert commands, and the current continuation expecting

After Denesting:

```
(defname rangeproduct
 (create
  (script ()
   (is-request
    (:do (lo hi) :unserialized)
    (let-except
      ((v0
        (let ((v1 (:= lo hi)))
          (if v1
            (then lo)
            (else (let ((v2 (let ((v3 (:- hi 1)))
                              (:= lo v3))))
                    (if v2
                      (then (:* lo hi))
                      (else
                       (let ((average (let ((v4 (:+ lo hi)))
                                        (floor v4 2))))
                         (let ((v5 (rangeproduct lo average))
                               (v6 (let ((v7 (:+ average 1)))
                                     (rangeproduct v7 hi))))
                           (:* v5 v6)))))))))))
      (except-when
       ((some-error (&rest args))
        (complain-to* customer (some-error reply-keyword args))))
      (reply-to customer (reply-keyword v0)))))))
```

Figure 7-4: RangeProduct after denesting (actually done as part of parsing).

the value returned from the expression (expr-pnode, inode-body, cont). Right now, the current parsed command is the *let-except* in the *is-request* body. The current inode body is the body of the handler we just created in the top node, and there is no continuation yet.

3. When we encounter the *let-except*, we generate a new continuation. In our example, the new continuation is represented by the bottom node in the graph. We also give this continuation a reply handler based on the identifier being bound, in this case v0. The *body* of the *let* is recursively processed with the new continuation as the current inode body. (I will ignore complaint handling for this example.)

4. Since they have been denested, commands, such as *reply-to*, are just inserted into the current inode body. Thus, when we process the *reply-to* in the body of the *let-except*, we just insert the *reply-to* into the current inode body, which is the new continuation. We can see the added *reply-to* command (abbreviated reply v0) in the body of the bottom node of the graph.

5. We recursively process the expressions in the *arms* of the *let-except* with the same current inode body as when we entered the *let* (the handler body for the top node), but with the new continuation created for this *let* (the bottom node) as the current continuation, since it

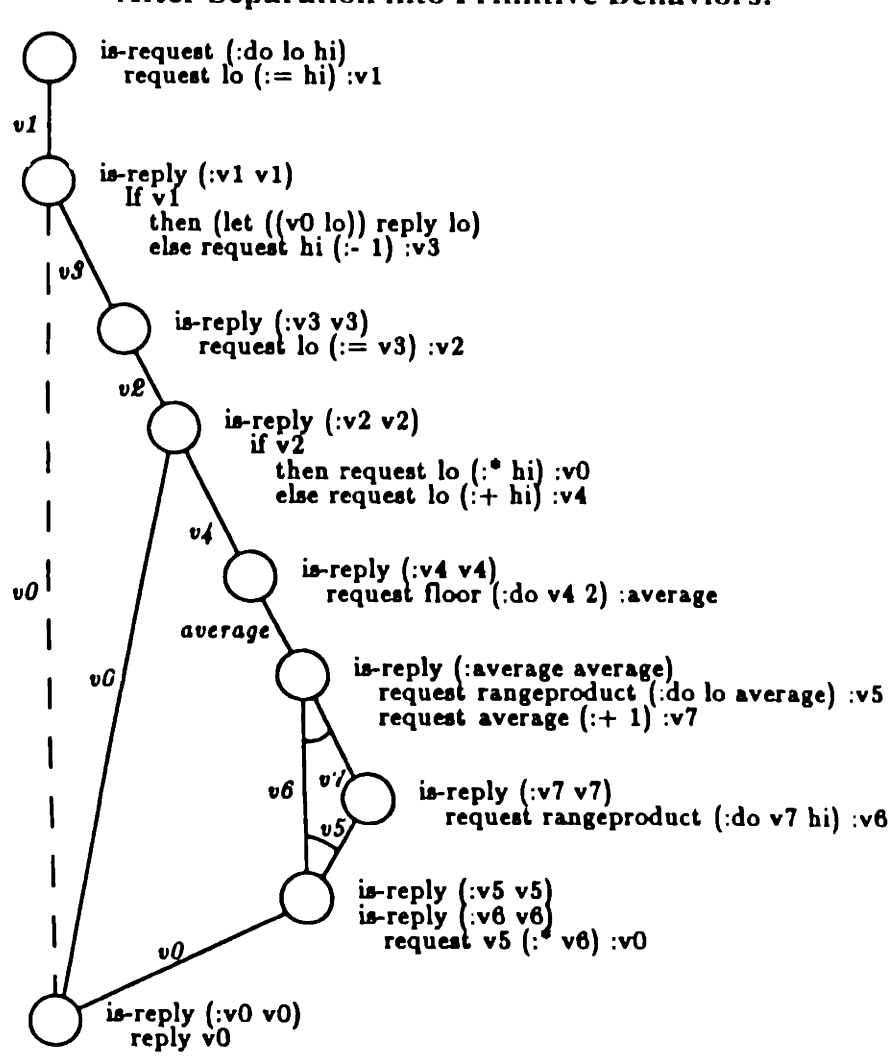


Figure 7-5: RangeProduct's graph of primitive behaviors

represents the behavior that will expect the values from the arm expressions.

6. The first thing we find in the arm of the *let-except* is another *let*, so another continuation is created, this time represented by the second node from the top in the graph. This *let* binds the value of v1, so we give the continuation a reply handler which expects a reply for :v1. To process the arm of this *let* we use this new continuation as the continuation, but we still keep the same current body, the first node.

7. When we encounter an ask expression, we add a request command to the current inode body; the customer of the request is the current continuation. Thus the ask expression (:= lo hi) produces the request command in the leading node, and the customer of this request is represented by the current continuation, which in this case is the second node.

8. To process the body of this *let*, the current inode body is the continuation generated for this *let* (the second node). The current continuation is the same as for the entire *let* expression (the bottom node), since the value of the *let* expression is the value of its body.

9. When we encounter an *if*, an *if* command is added to the current inode body, and two new inode bodies are created for the *then* and *else* branches. Then the *then* and *else* parse structures are recursively processed, inserting new commands into the new *then* and *else* inode bodies. In our example, the (*if* v1...) is added to the second node, and the expressions in the branches of the denested *if* are icoded into the branches of the *if* inode with the bottom node still as the continuation.

10. In the *then* branch of the *if*, we find a special situation. The expression is just the identifier lo, while the code to process it is in the continuation. In this case the *continuation pulling* optimization allows us to copy the body of the continuation into the current inode body. Thus we can pull the *reply-to* command from the body of the bottom continuation into the *then* arm of the *let*.

11. Most of the rest of processing follows the same lines as what we've done in previous steps for processing *let* expressions, *if* expressions, and ask expressions. The only slightly different step is to process the *let* with two arms, binding v5 and v6. This is a parallel *let*, which just forms a joining continuation. Since two transactions must share the joining continuation, it must be created at the fork, and it must receive multiple reply handlers.

I have left out many of the details, but I hope the basic idea of the separation phase is apparent from this little exercise. If we were to create Pract code directly from the results of separating the code into primitive behaviors, the product would be something like the code in figure 7-6 (omitting complaint handling). Several parts of the behavior are still missing: The behaviors don't declare the local acquaintances and constant acquaintances used; the behaviors are not connected with expressions for creating continuations or by commands updating to become the next continuation; and therefore no customer can be specified for each of the requests. The only requests which have their customer filled in are the requests in cont5-behavior and cont6-behavior which produce the values for the joining customer; the identifier (v8) has already been set up to bind the shared joining continuation at the fork at cont5-behavior. However, the identifier has not yet been bound, and the synchronization necessary at the joining customer for the parallel *let* has not been added.

7.2.4 Optimizing Tail Recursion

Since cont8-behavior merely forwards any response it receives, the tail recursion optimization can be made, eliminating the need for this behavior. If this optimization is made, cont3-behavior and cont7-behavior specify that the results of the ask expression are to be sent directly to the customer, as in figure 7-7.

7.2.5 Collecting Identifiers

The first step in taking care of declaring the local and constant acquaintance identifiers is to collect identifiers referenced within the body of each behavior into a set associated with the behavior itself. Recall that the parsing phase uniquely identified all identifiers and classified them according to type. This information is now used in collecting identifiers. All identifiers which were locally bound in the

Separated Primitive Behaviors

```
(defequate rangeproduct-behavior
  (script (()()
          (is-request ((:do lo hi) :unserialized)
            (request lo (:= hi) sponsor <??> :v1)))))
(defequate cont1-behavior
  (script (()()
          (is-reply (:v1 v1)
            (if v1 (then (let ((v0 lo)) (reply-to customer (reply-keyword v0))))
                   (else (request hi (:- 1) sponsor <??> :v3)))))))
(defequate cont2-behavior
  (script (()()
          (is-reply (:v3 v3)
            (request lo (:= v3) sponsor <??> :v2)))))
(defequate cont3-behavior
  (script (()()
          (is-reply (:v2 v2)
            (if v2 (then (request lo (:* hi) sponsor <??> :v0))
                   (else (request lo (:+ hi) sponsor <??> :v4)))))))
(defequate cont4-behavior
  (script (()()
          (is-reply (:v4 v4)
            (request floor (:do v4 2) sponsor <??> :average)))))
(defequate cont5-behavior
  (script (()()
          (is-reply (:average average)
            (let ((v8 <??>))
              (request rangeproduct (:do lo average) sponsor v8 :v5)
              (request average (:+ 1) sponsor <??> :v7))))))
(defequate cont6-behavior
  (script (()()
          (is-reply (:v7 v7)
            (request rangeproduct (:do v7 hi) sponsor v8 :v6)))))
(defequate cont7-behavior
  (script (()()
          (is-reply (:v5 v5)
            (request v5 (:* v6) sponsor <??> :v0))
          (is-reply (:v6 v6)
            (request v5 (:* v6) sponsor <??> :v0)))))
(defequate cont8-behavior
  (script (()()
          (is-reply (:v0 v0)
            (reply-to customer (reply-keyword v0))))))
```

Figure 7-6: RangeProduct after simply separating into behaviors

Acore script may need to be stored as local acquaintances of the continuations, so the local identifiers

```
...
(defequate cont3-behavior
  (script (()()
        (is-reply (:v2 v2)
         (if v2 (then (request lo (:* hi) sponsor customer reply-keyword))
             (else (request lo (:+ hi) sponsor <??> :v4)))))))
...
(defequate cont7-behavior
  (script (()()
        (is-reply (:v5 v5)
         (request v5 (:* v6) sponsor customer reply-keyword))
        (is-reply (:v6 v6)
         (request v5 (:* v6) sponsor customer reply-keyword)))))
```

Figure 7-7: Modifications for tail recursion optimization

referenced by the behavior are collected into a set. Similarly, free identifiers of the Acore script need to be collected as the sets of constant acquaintances of the continuations. Each primitive behavior also references the continuation behaviors connected immediately below it in the graph. Although the names are not visible in our Pract translation into figure 7-6, they are available from the graph of figure 7-5. These references are also constants, so the names of the continuation behaviors are also collected into the constant acquaintance set.

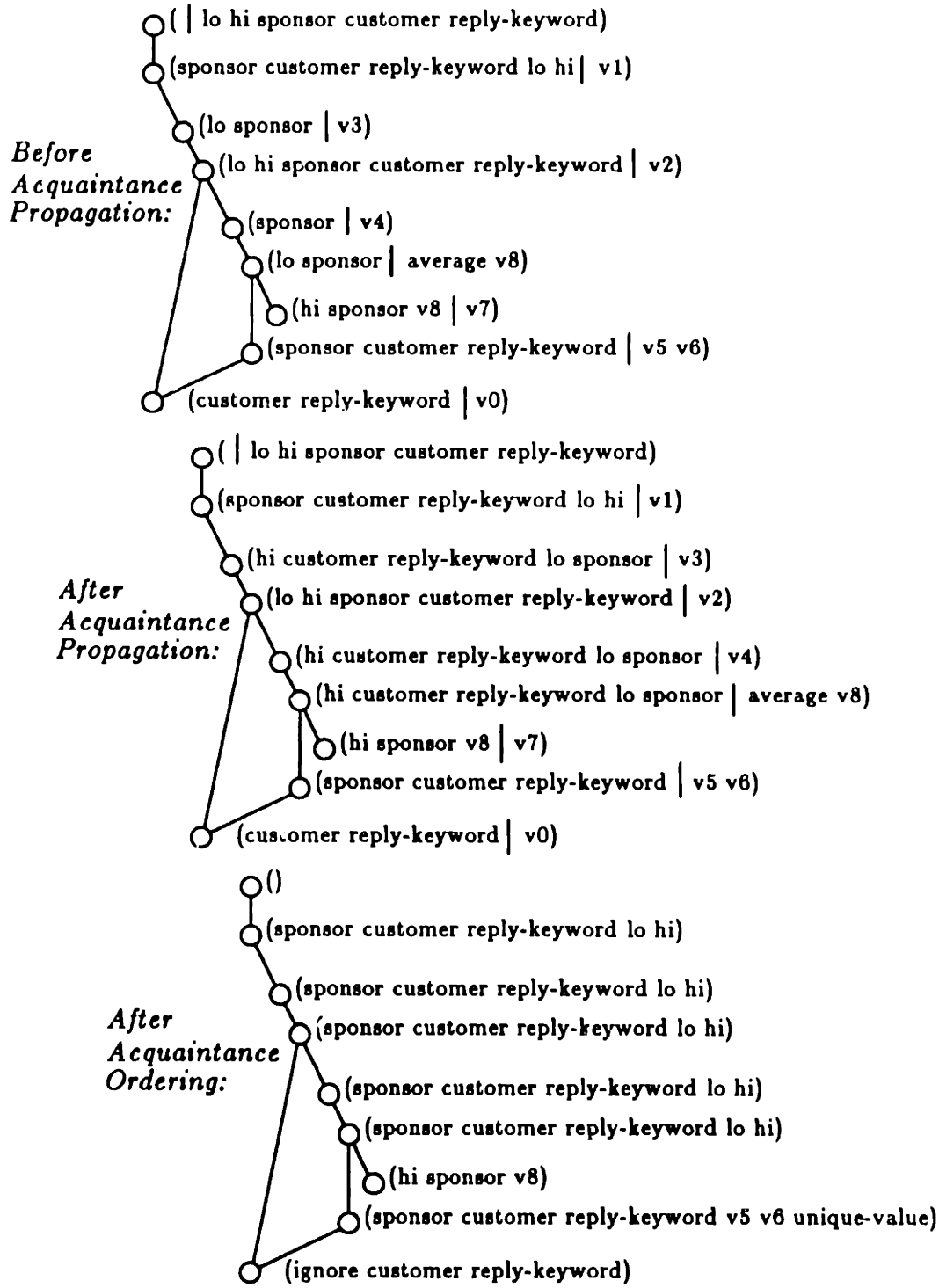
This collection is performed at the end of the separation phase. If we were to generate Pract code after the rangeproduct example was processed through identifier collection, it might look something like figure 7-8. First let's look at the constant identifiers. The only free identifiers in the original Acore script were floor and rangeproduct (which will be bound in the top level environment); all the other constant identifiers are names of continuation behaviors. Note that as a result of the tail recursion optimization, now none of the behaviors refer to cont8-behavior. Also, notice that cont6-behavior is an arm continuation for a parallel let, and the next continuation is a joining continuation. Since the joining continuation actor is already passed through as v8, it doesn't need to create or update to the joining continuation, and therefore it doesn't refer to the joining continuation behavior.

Now let's look at the local identifiers. The local identifiers are collected in two sets, a set of free local identifiers and a set of defined local identifiers. The set of defined local identifiers are those identifiers which are bound by that behavior, either by a message handler or by an interior Pract *let*. The set of free locals are any other local identifiers referenced; they must be passed to the behavior as acquaintances. For simplicity, we have indicated these two sets as the local acquaintance list separated by a vertical bar: (*free* | *defined*).

Behaviors with Identifiers Collected

```
(defequate rangeproduct-behavior
 (script (( | lo hi sponsor) (cont1-behavior)
         (is-request ((:do lo hi) :unserialized)
          (request lo (:= hi) sponsor ?? :v1)))))
(defequate cont1-behavior
 (script ((sponsor customer reply-keyword lo hi | v1) (cont2-behavior)
         (is-reply (:v1 v1)
          (if v1 (then (let ((v0 lo)) (reply-to customer (reply-keyword v0))))
              (else (request hi (:- 1) sponsor ?? :v3)))))))
(defequate cont2-behavior
 (script ((lo sponsor | v3) (cont3-behavior)
         (is-reply (:v3 v3)
          (request lo (:= v3) sponsor ?? :v2)))))
(defequate cont3-behavior
 (script ((lo hi sponsor customer reply-keyword | v2) (cont4-behavior)
         (is-reply (:v2 v2)
          (if v2 (then (request lo (:* hi) sponsor customer reply-keyword))
              (else (request lo (:+ hi) sponsor ?? :v4)))))))
(defequate cont4-behavior
 (script ((sponsor | v4) (floor cont5-behavior)
         (is-reply (:v4 v4)
          (request floor (:do v4 2) sponsor ?? :average)))))
(defequate cont5-behavior
 (script ((lo sponsor | average v8)
         (rangeproduct cont-7-behavior cont6-behavior)
         (is-reply (:average average)
          (let ((v8 ??))
           (request rangeproduct (:do lo average) sponsor v8 :v5)
           (request average (:+ 1) sponsor ?? :v7))))))
(defequate cont6-behavior
 (script ((hi sponsor v8 | v7) (rangeproduct)
         (is-reply (:v7 v7)
          (request rangeproduct (:do v7 hi) sponsor v8 :v6)))))
(defequate cont7-behavior
 (script ((sponsor customer reply-keyword | v5 v6) ()
         (is-reply (:v5 v5)
          (request v5 (:* v6) sponsor customer reply-keyword))
         (is-reply (:v6 v6)
          (request v5 (:* v6) sponsor customer reply-keyword)))))
(defequate cont8-behavior
 (script ((customer reply-keyword | v0) ()
         (is-reply (:v0 v0)
          (reply-to customer (reply-keyword v0))))))
```

Figure 7-8: RangeProduct after collecting each behavior's identifiers (free-locals | defined-locals)



Behavior references: (free locals | bound locals)

Figure 7-9: Propagating and ordering references in behaviors of RangeProduct

7.2.6 Propagating Acquaintances

This points out one of the problems tackled by the next stage: making sure that each behavior passes the references needed by subsequent behaviors. For example, cont3-behavior needs hi, customer, and reply-keyword, but cont2-behavior doesn't have these as acquaintances and can't pass them to it. Therefore, the *traverse DAG* stage starts at the leading behavior and walks a depth first spanning tree (actually all edges) of the directed acyclic graph of primitive behaviors. When it retreats backward over each line, it propagates local references back from the leaves towards the root. At each node it adds the references needed by following continuations to the set needed by the current behavior, except for the references which are bound by the current behavior. Thus, every local reference will be present as an acquaintance in every behavior between the point where it was defined and the last behaviors where it is used, following any path of behavior transitions through the intermediate behaviors.

To see how this works, let's suppose for a moment that we were unable to make the optimization which eliminated cont8-behavior. Then the graph of behavior dependencies with the local acquaintance lists as we have collected them up to this point can be described by the top graph of figure 7-9. The traverse DAG phase walks this graph depth first, propagating identifiers as it retreats from the leaves towards the root, producing the sets as indicated in the middle graph. This guarantees that each behavior has the local references needed to create the following behavior. As long as the walk is depth first, every edge of the DAG need be traversed only once; even though there may be multiple paths to a node, once all the identifiers from the nodes below it have been propagated up to it, there is no reason to explore those nodes again.

7.2.7 Connecting Behaviors

Now that each primitive behavior has identifiers for all the references it uses and all the references it needs to pass on to subsequent behaviors, the next step is to go through the behaviors and add the code to connect the behaviors: creating new continuations where needed, reusing continuations by updating where possible, filling in the customers for requests with the created or updated continuations. This phase works its way from the leading behavior, exploring the tree of commands in each handler. As references continuations are found in requests and at forks for parallel lets, new commands are added to connect the behaviors and the references are filled in.

For each of these connections, the decision must be made whether to reuse the current continuation with an update or to create a new continuation. Where there are concurrent forks, at most one of the branches may reuse the current continuation; for alternative forks resulting from a decision, either or both of the branches may reuse the current continuation. Since reusing a continuation is more efficient, the basic strategy is to reuse the continuation whenever possible, i.e. whenever it hasn't already be taken by another concurrent branch.

Connections involving a joining continuation also need to initialize it correctly. In section 6.4 we saw that our method for implementing joining continuations required some additional acquaintances for

storing the returned values until the last value is returned, and one more for storing the unique value. These acquaintances must be added to the behavior of the joining continuation, and must be initialized by the actor creating or becoming the joining continuation. Thus, in the rangeproduct example, cont7-behavior will require three additional acquaintances (v5, v6, and unique-value), and the code which is added to cont5-behavior to create or become the joining cont7-behavior needs to initialize these acquaintances to the unique value.

7.2.8 Ordering Acquaintances

An optimization can be made to help make the updates efficient. Looking at the acquaintance lists of the middle graph of figure 7-9, you may notice that the identifiers are not in any particular order. If left as they are, update commands connecting the behaviors will need to do some extra work just to rearrange the acquaintances of the continuation as it takes on subsequent continuation behaviors. Therefore, an important part of the behavior connecting phase is to determine the ordering and placement of the acquaintances of the primitive behaviors.

The acquaintance propagation phase worked from the leaves of the continuation DAG towards the root (the leading behavior), propagating the identifiers needed by later continuations towards the behaviors where they are bound. Now the behavior connecting phase works from the root towards the leaves, deciding which behavior transitions can be made through updates, and ordering the acquaintances of the updated-to behavior to match the positions of the updated-from behavior has much as possible. To avoid shuffling the acquaintance positions when a value is not needed by a subsequent behavior, an ignored identifier may be inserted to serve as a placeholder. Ignored positions may also be added at the end of the acquaintance list to hold places for subsequent behaviors which need additional positions to hold intermediate values (these need not be explicitly added to the behaviors). Since the update command does not change the size of the actor, this means some extra positions may need to be included when the continuation is created.

The result of ordering acquaintances on our simple rangeproduct example is shown in the bottom graph of figure 7-9. After propagating acquaintances, the sets indicating where identifiers are bound are no longer needed and have been omitted. Notice that the sixth continuation, which is the arm continuation of the parallel let, will be created, so it needn't follow the acquaintance ordering of the rest of the continuations. Acquaintances have been added to the joining continuation; v5 and v6 take the place of hi and lo which are no longer needed. The last continuation has an ignore place holder where sponsor used to be. The ignored positions at the end of the acquaintance lists are omitted; they add no positional information.

7.2.9 Ordering Behaviors

Before we can generate the Pract code, we need to make sure that the scripts are defined in a way so that every script is defined before it is referenced by another script. The easiest way to do this and still

make the whole definition an expression is, as we saw in section 6.7, to linearize the graph into a series of nested Pract let expressions, each defining a single primitive script. To accomplish this linearization, a total depth first traversal of the continuation graph is made. Every time a continuation is encountered, it is added to the top of a stack; at the end of the process, the duplicates are removed. (This traversal is actually combined with the acquaintance propagating traversal.)

7.2.10 Generating Pract

Once we've filled in the commands to connect the primitive behaviors, the only thing left to do is produce the Pract code. For the most part, generating Pract is straightforward; the behavior separation phase produced a structure which follows the command structure of Pract behaviors, and later phases have just filled it out with identifier information and commands to connect the behaviors. The only thing we've left out is producing the handlers and tests for the joining continuation, but this is just a matter of filling in a pattern for the handlers with the test and the continuation body.

The completed Pract rangeproduct is shown in figure 7-10; I have left it as a series of *defequate*'s rather than as nested *let*'s for readability and to make it easy to compare with the previous versions. The last continuation is gone since the tail recursion optimizations rendered it unnecessary. All the updates and creates for connecting the behaviors have been filled in, and the requests have their customers filled in. The first continuation actor is created with an extra acquaintance position filled with 'ignore so that there will be enough positions for the joining customer. At the fork for the parallel *let* in cont3-behavior, the current continuation is set up as the joining continuation, and initialized with the unique value in the positions for storing the values from the concurrent arms. It's done!

Now, with a good idea of what the compilation process is like, we can better understand the relevance of the details as we look into each of the compilation stages in more depth.

7.3 Macroexpansion Issues

The first step in the compilation of Acore is the macroexpansion phase. Acore source code is manipulated in the form of list structure, so macros transform list structure representing a syntactic abstraction into list structure expressing the equivalent behavior in Acore. Since Acore uses an expansion passing style macro facility, there is an expander function for every special form in the language. Most of these are straightforward, but there are some interesting points concerning the expander function for *script* expressions and the shielding of locally bound names in *let*, handlers, and *script*.

I described the design of Acore macroexpansion facility in section 4.11.3. There, we discussed the merits of *expansion passing style* macros over traditional Lisp macros, and showed how expansion style macros are both more powerful and more modular. The modularity comes from separating the processing of each macro and special form into its own expander function. The power comes from giving each macro's expander function complete control over what subforms are expanded and what is returned from

Completed RangeProduct

```
(defequate rangeproduct-behavior
 (script
  (() (cont1-behavior)
   (is-request ((:do lo hi) :unserialized)
    (request lo (:= hi) sponsor
     (create cont1-behavior sponsor customer reply-keyword lo hi 'ignore)
     :v1)))))
(defequate cont1-behavior
 (script ((sponsor customer reply-keyword lo hi) (cont2-behavior)
       (is-reply (:v1 v1)
        (if v1 (then (let ((v0 lo)) (reply-to customer (reply-keyword v0))))
            (else (update self (0 cont2-behavior))
                (request hi (:- 1) sponsor self :v3)))))))
(defequate cont2-behavior
 (script ((sponsor customer reply-keyword lo hi) (cont3-behavior)
       (is-reply (:v3 v3)
        (update self (0 cont3-behavior))
        (request lo (:= v3) sponsor self :v2)))))
(defequate cont3-behavior
 (script ((sponsor customer reply-keyword lo hi) (cont4-behavior)
       (is-reply (:v2 v2)
        (if v2 (then (request lo (:* hi) sponsor customer reply-keyword))
            (else (update self (0 cont4-behavior))
                (request lo (:+ hi) sponsor self :v4)))))))
(defequate cont4-behavior
 (script ((sponsor customer reply-keyword lo hi) (floor cont5-behavior)
       (is-reply (:v4 v4)
        (update self (0 cont5-behavior))
        (request floor (:do v4 2) sponsor self :average)))))
(defequate cont5-behavior
 (script ((sponsor customer reply-keyword lo hi)
       (rangeproduct cont-7-behavior cont6-behavior)
       (is-reply (:average average)
        (let ((v8 self))
         (let ((unique-value (create null-script)))
          (update self (0 cont-7-behavior) (4 unique-value)
                (5 unique-value) (6 unique-value)))
         (request rangeproduct (:do lo average) sponsor v8 :v5)
         (request average (:+ 1) sponsor
          (create cont6-behavior v8 sponsor hi) :v7))))))
(defequate cont6-behavior
 (script ((hi sponsor v8) (rangeproduct)
       (is-reply (:v7 v7)
        (request rangeproduct (:do v7 hi) sponsor v8 :v6)))))
```

(continued on next page)

(continued from previous page)

```
(defequate cont7-behavior
 (script ((sponsor customer reply-keyword v5 v6 unique-value) ()
        (is-reply (:v5 v5)
         (if (or (== unique-value ':abort) (== v6 unique-value))
          (then (update self (4 v5)))
          (else (request v5 (:* v6) sponsor customer reply-keyword))))
        (is-reply (:v6 v6)
         (if (or (== unique-value ':abort) (== v5 unique-value))
          (then (update self (5 v6)))
          (else (request v5 (:* v6) sponsor customer reply-keyword)))))))
```

Figure 7-10: RangeProduct Compiled to Pract

---

the macroexpansion. This control is achieved by calling the macro's specialized expander function with the form to be expanded *and* a general macroexpansion function with which to process subforms. The expander function has complete responsibility for returning a fully macroexpanded form, since no further macroexpansion is done on the form returned. Since this power and modularity is important in a language designed partially for experimenting with syntactic abstractions for concurrent programming, I chose to base the Acore macro expansion facility on the expansion passing style.

As a result of choosing the expansion passing style, every type of special form in Acore must have an expander function which expands the macros inside forms of that type. Most of these are fairly straight forward, and just apply the general macroexpansion function (passed as a parameter) to the subexpressions supplied by the programmer. For example, the *if* expander function expands the test expression and the expressions in the *then* and *else* bodies, and returns an *if* with the macroexpanded subexpressions substituted. Schematically, the *if* macro expander function takes list structure of the form

(if *expr1* (then *expr2*) (else *expr3*))

and returns list structure of the form

(if *expr1** (then *expr2**) (else *expr3**))

where *expr1*, *expr2*, and *expr3* are the expressions supplied by the programmer and *expr1**, *expr2**, and *expr3** are the results of macroexpanding each of them.

Not all the parts supplied by the programmer should be expanded. For example, the parameter to *quote* should never be expanded; in fact the expander for *quote* just returns the form unchanged. Symbols which are not evaluated should not be expanded, so the identifier bindings in a *let* or the parameter lists of message handlers should not be expanded.

The only special form expander which is not as straightforward is performed for the *script* expression. Recall that the default message handlers accept request messages and return the value of the expression in their body. These handlers have the form:

```
(script (acquaintances)
  ...
  ((:keyword (parameters...) options...)
   expression-body...)
  ...)
```

The *expression body* evaluates to the value returned in response to the request. If a complaint is generated instead, it should be forwarded to the customer of the request. As we discussed in section 6.6, this behavior is equivalent to the behavior produced by the following handler:

```
(script ()
  ...
  (is-request (:keyword (parameters...) options...)
   (let-except ((value (let () expression body...)))
               (except-when
                ((some-error (&rest parameters))
                 (complain-to* customer
                  (some-error reply-keyword parameters))))
    (reply-to customer (reply-keyword value))))
  ...)
```

The script expander function takes care of this transformation, simplifying the compiler. This transformation is the source of the *let-except* and complaint handler which appeared in figure 7-3 (page 98) after we expanded our rangeproduct. The script expander function also takes care of transforming the script expression into a form which encapsulates the actual script in a guardian, as was described in section 6.7.

The macroexpansion functions supplied for Acore special forms perform one other function which may not be obvious. As we discussed in section 4.11.3, it is a good idea to apply the idea of lexical scoping to macros names as well as bound identifiers, so that expressions referring to locally bound identifiers aren't expanded by mistake. Therefore the macroexpansion functions for the forms which bind names, i.e. *let* (identifiers bound in arms), *script* (acquaintances), the message handlers *is-request*, *is-reply*, and *is-complaint* (message parameters), and *let-except* (complaint handler parameters), all protect the locally bound identifiers. The simplest way to do this is to extend the macro environment with identifiers bound the ask expression expander, so that if they are found at the head of a form, the form will be treated as an ask expression.

Speaking of mistakes, sometimes an error may occur in a macro either because of a mistake in the macro function or because the source code is of the wrong form. Since Acore is designed to be extended for developing and experimenting with concurrent languages, macro forms should not be second class when it comes to error reporting. Therefore, the current implementation catches errors occurring in calls to expander functions and reports them in the same way as compiler errors.

Since the Acore compiler is currently written in Lisp, and the macroexpansion facility is a preprocessor for the compiler, it is also written in Lisp. This means that although *DefExpander* and *DefMacro* forms are given, the bodies of these forms must be written in Lisp. Thus, the macroexpansion facility makes a list structure to list structure transformation, the output of which is passed to the parsing phase.

7.4 Parsing Issues

The parsing phase takes care of several chores for the compiler:

- Syntax checking.
- Parsing forms into structures onto which type information can be attached.
- Identifying which expressions are *continued* and contain ask expressions.
- *Denesting* forms with continued subexpressions into equivalent *let* forms.
- Uniquely identifying and classifying identifiers.
- Associating ask expressions with sponsors.

Parsing proceeds by finding the parser for an expression and calling the parser with the expression and the *parsing environment*. The parser checks that the expression has the correct form, extends the environment if necessary, and recursively parses any subexpressions of the form with the extended environment. A parsed node is then constructed using the parsed subexpressions. If any of the subexpressions are continued, a denested node may be created, in which a *let* first binds the values of the continued subexpressions before performing the action. At the leaves of the parse tree are the identifier nodes; a separate node identifies each uniquely bound identifier and stores its type in the current context (e.g. free, acquaintance, local to handler) as well as its name. In this section we'll look at the issues involved in each of these tasks.

7.4.1 Syntax Checking

Syntax checking is generally straightforward. In order to parse a form, it must be recognizable and its subexpressions must fit the pattern for the form. For example, and *if* expression requires a test expression, a *then* clause, and an *else* clause:

```
(if test-expression
  (then expression-body...)
  (else expression-body...))
```

The only interesting point about syntax checking is the generation of error messages. The current implementation keeps track of contextual information on the environment stack, e.g. what definition, what handler of the definition, and what command inside the handler is currently being compiled, so that whenever an error message is generated this information can be included. It is conceivable that future compilers may be further integrated with the editing system so that the source code which generated the syntax error can be automatically located for the programmer, but keeping track of such pointers may unduly complicate the macro system and macro writing.

7.4.2 Constructing Pnodes

A major purpose of any parsing stage is to transform the input source code into a tree of structures (*pnodes*) which may be easily manipulated by later stages of the compiler. To perform this transfor

mation, the parser must recognize the various types of source forms and destructure them into their constituent parts for further parsing. Some of this information may depend on context; for example whether an identifier is a free or local identifier depends on whether it is locally bound in the current context. To store the information thus gained by this recognition and destructuring process, the parse stage constructs a tree of nodes with easily identifiable types and easily accessible substructures. Additional information may also be added to help later stages of the compilation: for example, more general classes such as whether a particular node represents a command or an expression. We will see that other kinds of useful information is found and stored during the parse in the following sections.

7.4.3 Identifying Continued Expressions

One kind of useful information is whether or not an expression contains any ask expressions. Expressions which do not contain ask expressions may be translated directly into Pract expressions. Expressions which do contain ask expressions will require generating a continuation — we call these *continued* expressions. By definition, ask expressions are continued expressions. Other parsers check to see if any of their subexpressions are continued, and the pnode created is marked accordingly.

7.4.4 Denesting Continued Subexpressions

The only significant transformation done by the parsing phase is to *denest* continued subexpressions. As was shown in our rangeproduct example, this involves pulling the continued subexpression out from the parameter position of the expression which contains it, binding the value in a *let*, and substituting the bound name in the containing expression. The *let* source form is not actually constructed; instead, the pnode for the *let* is created directly. In order to perform this step, each parser must be able to identify whether or not its subexpressions are continued; hence the previous section. It is possible to denest all subexpressions, but this would simply produce extra *let* expressions at the Pract level, which may or may not reduce the efficiency of the compiled code depending on the sophistication of the Pract compiler. However, it does produce more nodes to be processed by the Acore compiler, so I made the decision to denest only continued expressions.

Denesting applies to all expressions and commands which are evaluated in an applicative order. This includes almost everything except *script*, *if*, and *let*. The expressions and commands within the handlers of a *script* expression are closed within the script, and thus should not be pulled outside the script. However, expressions within commands inside the handlers are denested inside the handlers during the parse of the script. The test expression for an *if* may be denested, but the expressions within the *then* and *else* bodies should not be pulled outside the if since they may not be evaluated depending on the value of the test. For example, an *if* expression may be denested as in figure 7-11. Subexpressions are not pulled outside a *let* — in the case of the arms, it would serve no purpose, and the body expressions cannot be pulled outside the environment set up by the arms. While subexpressions in certain locations are not pulled out of those locations, the subexpressions themselves are denested as necessary (figure 7-12). As a result of denesting, every ask expression is located in the arm of a *let*.

WRONG: else expression gets evaluated without regard to the value of (:= 1 n)>.

```
(if (:>= 1 n) (then 1) (else (:* n (f (:- 1 n)))))
→
(let ((v1 (:>= 1 n))
      (v2 (:* n (f (:- 1 n)))))
  (if v1 (then 1) (else v2)))
```

RIGHT:

```
(if (:>= 1 n) (then 1) (else (:* n (f (:- 1 n)))))
→
(let ((v1 (:>= 1 n)))
  (if v1 (then 1) (else (:* n (f (:- 1 n))))))
```

Note: Only the if denesting is shown; (: n (f (:- 1 n))) has not been denested.*

Figure 7-11: Denesting of *if*.

Denesting an else expression in place:

```
(if (:>= 1 n) (then 1) (else (:* n (f (:- 1 n)))))
→
(let ((v1 (:>= 1 n)))
  (if v1 (then 1) (else (let ((v2 (let ((v3 (:- 1 n)))
                                    (f v3))))
                          (:* n v2))))))
```

Denesting a let expression in place:

```
(let ((average (floor (:+ hi lo) 2)))
  (:* (rangeproduct lo average)
      (rangeproduct (:+ average 1) hi)))
→
(let ((average (let ((v1 (:+ hi lo)))
                 (floor v1 2))))
  (let ((v2 (rangeproduct lo average))
        (v3 (let ((v4 (:+ average 1)))
              (rangeproduct v4 hi))))
    (:* v2 v3)))
```

Figure 7-12: Denesting in place

While writing this chapter I noticed a strong correspondence which may help readers who are familiar with work on Scheme compilers better understand this part of compilation into primitive actors. The denesting operation is very similar to the conversion to continuation passing style performed by the Rabbit Scheme compiler [Steele 78], also described earlier by Steele [Steele 76]. In both cases, all expressions requiring continuations — applications in Scheme and ask expressions in Acore — are

separately bound to variables. In Acore they are bound with *let* while in Scheme they are bound with a *lambda*, but the two forms are equivalent — in fact many Scheme implementations translate each *let*:

```
(let ((var1 expr1) (var2 expr2) ...)
  <body>)
```

into the equivalent *lambda* form:

```
((lambda (var1 var2 ...)
   <body>)
 expr1 expr2 ...)
```

However, there is an important difference. The conversion to continuation passing style performed on Scheme programs introduces a sequential evaluation order on the arguments to an application, serializing the subexpressions into a sequence of continuations. Since such expressions are concurrent in Acore, the denesting operation described here preserves the parallel structure of adjacent subexpressions. Thus, for example, in figure 7-12, the expressions for v2 and v3 may still be evaluated concurrently.

Aside from concurrency, the two forms are compiled in a similar manner. The applications of Scheme are compiled into jumps, while ask expressions in Acore are compiled into message passes. There are two types of applications in Scheme: applications which supply a continuation while calling a function, and applications of continuations; these correspond to request messages to leading actors and reply messages to continuation actors. This similarity shouldn't be surprising since the Scheme work was inspired in part by Carl Hewitt's early work with Actors, but I had not noticed it earlier.

7.4.5 Parsing Identifiers

The example compilation of rangeproduct showed that a large part of the compiler's work deals with collecting and propagating references. Therefore an important task of the parsing phase is to identify each symbol according to where it is bound, and classify them according to the type of reference. After describing how this is done, we'll take a brief look at two subtle issues which must be dealt with to parse identifiers correctly.

Parsing identifiers is performed by passing an environment throughout the parsing phase; the parsers for forms which bind symbols extend the environment with the symbol and its pnode, and the forms in the scope of that binding are parsed with the extended environment. Thus, parsing an identifier just involves looking up the identifier in the environment; if it is not found, it must be a new free identifier, and is added at the outermost level.

Identifiers are classified into three types: local names, acquaintance names, and free names. Acquaintances are distinguished from identifiers bound locally from the message or in a *let* since only acquaintances may be updated with a *ready* command; the acquaintance name pnode also stores the acquaintance's position so the *ready* may later be compiled into a Pract *update* command. Because identifiers are referentially transparent, free names are constants within a behavior and their values become part of the script. Therefore free names do not need to be passed between continuations, as we saw in the rangeproduct example, while references to acquaintances and locally bound identifiers must

be so propagated.

One small problem occurs if a *let* binds a name already locally bound in its surrounding scope. Since both values may need to be propagated by the same continuations, a name conflict can occur in the acquaintances of the Pract scripts. To resolve this problem, if this situation occurs, the parser renames the inner bound identifier to prevent the conflict. Each identifier pnode stores the name of the pnode which will be produced on output; the renaming is performed by adding a '~' (representing "prime") to the output form of the conflicting symbol (e.g customer becomes customer~). Thus if the conflicting symbol had also been primed, the new symbol would be double primed; i.e. arbitrary nesting is supported. Although this technique isn't completely general since it restricts programmers from using this method themselves, it does preserve the original name of symbol, which is helpful in debugging.

The other subtle issue in parsing identifiers is dealing with script boundaries. Identifiers which are free inside a *script* expression may be local outside the *script* if the *script* expression is nested within another *script*. Since local names and free names have different types, the same identifier pnode cannot represent the reference in both contexts. For example, in the expression:

```
(script ()
 ((:do (n))
  (let ((n2 (:* n 2)))
    (:create (script ()
               ((:n ()) n)
               ((:n2 ()) n2))))))
```

the outer script describes a behavior which returns an actor storing the values of n and n2. Let's concentrate just on n. In the inner script, n is a free reference. In the context outside the script, however, n is a locally bound name, and in fact, the inner script is the only reference to n in the continuation which receives the value of (:* n 2). Therefore, the reference to n made by the inner script must be recorded as a reference to the local identifier n in the outer script so that the reference will be propagated to the continuation correctly. To deal with this issue, when the parser is looking up an identifier, if it crosses a script boundary, the pnode returned by parsing the identifier in the outside environment is stored as a reference made by the *script* expression, and a new free name pnode is created and stored at the script boundary and returned for use within the script expression.

7.4.6 Associating Sponsorship

The final task of the parse phase is to associate a sponsor with every ask expression. Sponsorship is controlled by context; by default, the sponsor of an incoming request sponsors all the expressions within the handler receiving that request. However, as you may have noticed in the rangeproduct example, references to the identifier to which the sponsor is bound must also be propagated. Therefore, every ask expression must be associated with a sponsor.

To accomplish this, the parsers for request handlers and the *with-sponsor* form extend the environment with a special sponsor binding to store the identifier to which the current sponsor is bound.

Thus, whenever an ask expression is parsed, the sponsor is looked up in the environment and stored in the ask pnode. Because the compiler may need to generate ask expressions in certain circumstances, the sponsor is looked up and stored in pnodes which have expression bodies as well. We'll see what these circumstances are later.

Now that the parse phase has checked the syntax, identified and denested continued subexpressions, identified and classified identifiers, and associated sponsors with each ask expression, the tree of pnodes returned is ready to be separated into primitive behaviors by the following phase.

7.5 Separating into Primitive Behaviors

The primary purpose of the separation phase is to divide the actions specified by the parse tree into the primitive leading and continuation behaviors. As we saw during the rangeproduct example, this is performed by walking the parse tree, creating the leading behavior *inode* from script pnodes and continuation inodes from the bodies of let pnodes, and filling in the bodies of these intermediate nodes with the appropriate commands. Finally, this phase also collects the identifiers referenced in each of the behaviors.

The denesting operation of the parse phase is an important preparation which makes this possible. As a result of denesting, the actions are encountered in the parse tree in the order they are to be performed by the behaviors, all ask expressions are separated into the arms of a let, and all continuations are represented by the body of a let. We now look into how the separation phase processes each type of pnode. For brevity I will use the term *inode* to represent the intermediate structures output by this phase; the process of transforming the parsing structures (pnodes) into inodes will be called *icoding*.

7.5.1 Icoding Scripts and Handlers

When the separation phase encounters a script pnode, a script inode representing the behavior of the leading actor is created and each of the script's handlers is processed and added to the script inode. To process a handler, a handler inode is created. The body of the handler pnode is then processed; the handler inode forms the current body to which commands will be added. The script and handler inodes will represent the leading behavior once the icoding process is through; this is the start of our behavior graph.

7.5.2 Icoding Commands

Commands represent no value, so icoding commands represents simply adding commands to the current handler body. Thus command icoders take only the parameters (pnode current-inode); the parsed commands are represented by the pnode, and the icoded commands are added to the current inode. We saw in section 6.6 that expression body handlers are equivalent to a command body handler which evaluates the expression body with complaint handling and replies with the value; we saw in section 7.3

that the macro expander for *script* expressions takes care of this transformation. Thus all handlers of the script have command bodies, and the commands in these bodies are icoded using this strategy.

7.5.2.1 Simple Commands

Simple commands are the easiest forms to icode. Because of the denesting done by the parse phase, simple commands for sending messages such as *request*, *reply-to*, or *ready* contain no ask expressions, so they may be simply inserted into the current command body. Any subexpressions must be simple expressions which can be compiled by Pract.

```
icode-command(simple-command current-body-{...})
    ;; add simple command to current body
    ;; produces: current-body-{simple-command + ...}
  (add-command simple-command current-body)
```

7.5.2.2 *If* commands

If commands are also straightforward to icode. An *if* command represents a decision choosing which of two command bodies to perform. The strategy is just to insert an *if* inode into the current command body, and icoding the *then* and *else* bodies by inserting commands into the *then* and *else* branches of the *if* inode. Since the test expression has been denested, the only problem is to icode the *then* and *else* bodies. Two inodes are created for the *then* and *else* branches, and as each pnode branch is icoded, commands are inserted into the respective body inode. Once this has completed, the inode bodies and primitive test expression are combined into an *if* inode and inserted into the current body.

```
icode-command(if-command-{test then-commands else-commands}
              current-body-{...})
  (let ((then-inode (make-then-body-inode))
        (else-inode (make-else-body-inode)))
    (icode-commands then-commands then-inode)
    (icode-commands else-commands else-inode)
    (let ((if-inode (make-if-inode test then-inode else-inode)))
      (add-command if-inode current-body)))
    ;; produces:
    ;; current-body-{if-inode{test
    ;;                        then-inode-{<icoded then-commands...>}
    ;;                        else-inode-{<icoded else-commands...>}}
    ;;              + ...}
```

7.5.2.3 *Let* commands

Let commands can be classified into three types: simple *let* commands which have zero arms with continued expressions, single *let* commands which have one arm containing a continued expression, and parallel *let* commands which have two or more arms with continued expressions.

Simple *let* commands are icoded by creating a *let* inode and then icoding the body by inserting commands into this inode. This is possible since the arms of the *let* can be compiled into Pract

expressions, so a Pract *let* will do. Finally, the *let* inode is inserted into the current body.

```
icode-command(let-command{simple-arms body-commands} current-body-{...})
 (let ((let-inode (make-let-inode simple-arms)))
  (icode-commands body-commands let-inode)
  (add-command let-inode current-body)
    ;; produces:
    ;; current-body-{let-inode{arms {<icoded body-commands>}} + ...}
```

The icode generated from single *let* commands must create a continuation behavior which expects the reply from the single ask expression. The pattern for the reply handler is derived from the pattern to which the results will be bound. Normally this is just a single identifier, but we saw in section 6.8.4 that this can easily be extended to multiple values. Thus the pattern is derived from the identifier or list of identifiers by prepending the keyword to be used. The keyword may be anonymously generated, but to help debugging, I have chosen to give the reply keyword the name of the variable being bound. For example, in figure 7-6 (page 103) we see that the handler which receives the value for average expects the keyword :average; the message returning this value will also contain the keyword :average, and will be displayed by debuggers such as Traveler (appendix B).

The body of the *let* represents what to do once the value of the expressions in the arms is known; thus the body of (the reply handler of) the continuation is generated by icoding the body of the *let*. Since the returned value is bound to the identifiers in the reply handler, these commands will be performed in a run-time environment extended with identifier and its value. The continued arm is icoded using the current inode body, the continuation, and the pattern; we will discuss how expressions are icoded later. The *let* itself adds no commands to the current body; what needs to be added to the current body depends upon the expressions in the arms. The continuation inode is connected to the current inode through the results of icoding the ask expressions in the arms; we will see how this works when I describe how ask expressions are icoded.

We will also see that the icoding the arms may require the results of icoding the body for certain optimizations, such as "pulling" the body of the continuation. We saw an example of this in the first continuation of rangeproduct in section 7.2.3. I will get to this continuation pulling operation later; I mention it here because it affects how the parts of a *let* are icoded. In the current implementation, this constraint is satisfied by icoding the body of the *let* before the arms.

```
icode-command(let-command{single-continued-arm body-commands} current-body-{...})
 (let ((handler-pattern (make-pattern (binding-pattern single-continued-arm))))
  (let ((cont-inode (make-continuation-inode handler-pattern))
   (icode-commands body-commands cont-inode)
   (icode-expression (binding-expression single-continued-arm)
                     current-body
                     cont-inode
                     handler-pattern)))
```

The icode generated from parallel *let* commands is similar to single *let* commands except that they

must set up a fork and build a joining continuation. Instead of a single reply pattern, the joining continuation inode keeps track of multiple reply patterns, but otherwise the patterns and reply keywords are generated in the same way. The body of the *let* represents what the joining continuation should do once all the replies have been received; therefore the body of the joining continuation is generated by icoding the body of the *let*. To create the join, multiple handlers will need to be created with the synchronizing checks we first saw on page 75 in figure 6-12. However, at the icoding stage only a single body is generated; it will be combined with the multiple handler patterns and checking only in the last stage of compilation.

To create the fork, multiple request commands will be generated by the multiple continued arms. However, the expressions in the arms need to share the joining continuation so that at the end of each arm they will return a value to the joining continuation, as we saw in on page 78 in figure 6-14. To accomplish this, a *let* inode is created to bind the value of the joining continuation, and the arm expressions are icoded with this inode as the current body. Thus the requests for the fork produced by this *let* are nested in the body of this *let*, as in cont5-behavior from figure 7-6:

```
(defequate cont5-behavior
 (script (()()
          (is-reply (:average average)
           (let ((v8 ??)) ;; let to bind shared joining customer
            (request rangeproduct (:do lo average) sponsor v8 :v5)
            (request average (:+ 1) sponsor ?? :v7))))))
```

For this reason, this *let* inode is sometimes called the *dispatching* inode. Thus the arms are icoded using the dispatching inode as the current body, the identifier for the joining continuation, and their individual patterns.

```
icode-command(let-command{parallel-continued-arms body-commands} current-body-{...})
 (let ((jcont-identifier (new-identifier)))
  (let ((dispatching-inode ;; nil now; fill in value while connecting behaviors
         (make-let-inode (make-binding jcont-identifier nil))))
   (add-command dispatching-inode current-body)
   (let ((jcont-inode (make-joining-continuation-inode
                       (mapcar binding-pattern parallel-continued-arms))))
    (icode-commands body-commands jcont-inode)
   (mapc (lambda (binding)
          (icode-expression (binding-expression binding) ;; pnode
                            dispatching-inode          ;; current-body
                            jcont-identifier           ;; continuation
                            (make-pattern (binding-pattern binding)))) ;; pattern
     parallel-continued-arms))))
```

7.5.3 Icoding Expressions

Expressions do represent a value, and therefore every expression icoded must have a continuation which expects the value. Thus, the parameters for icoding expressions are (pnode current-body continuation pattern). The continuation represents the customer of the final request which produces the value of the expression. The pattern indicates what reply keyword the continuation expects and therefore

must be supplied by the request.

7.5.3.1 Ask expressions and the tail recursion optimization

Ask expressions are the bridges between behaviors. They initiate a transaction, the request of which is sent by one behavior and the reply of which returns to the continuation behavior. Since they've been denested, icoding them is relatively simple: it is just a matter of creating a request inode with the correct continuation and the reply keyword from the pattern, and adding the inode to the current body. At this point, the only connection between the current inode body and the continuation inode may be through this request inode.

```
icode-expression(ask-expression{:keyword target sponsor parameters}
          current-body continuation pattern)
  (let ((request-inode (make-request-inode target keyword parameters
                        sponsor continuation (pattern-keyword pattern))))
    (add-command-inode request-inode current-body))
```

However, we can make an optimization, the tail recursion optimization I have mentioned several times earlier. If the continuation merely forwards all replies and complaints in the exact same form as they are received, then we can extract the customer and its reply keyword from the continuation, and substitute those directly for the customer and reply keyword, just as was shown on page 73 in figure 6-10. (This is one of the optimizations which requires that the body of a *let* be icoded before the arms so that the body of the continuation will be complete when the expressions are icoded.)

7.5.3.2 *If* expressions

If expressions are icoded in a manner very similar to how *if* commands were icoded. The only difference is that the value of the *if* is the value returned by either the *then* body or the *else* body. Therefore the two bodies must be icoded with the incoming continuation inode and pattern so that at the end of the body the value will be returned to the continuation.

```
icode-command(if-expression-{test then-expression-body else-expression-body}
          current-body continuation pattern)
  (let ((then-inode (make-then-body-inode))
        (else-inode (make-else-body-inode)))
    (icode-expression-body then-expression-body then-inode continuation pattern)
    (icode-expression-body else-expression-body else-inode continuation pattern)
    (let ((if-inode (make-if-inode test then-inode else-inode)))
      (add-command if-inode current-body)))
```

7.5.3.3 *Let* expressions

Let expression are also icoded very similarly to how *let* commands were icoded. The only difference is a *let* expression has a value, the value of its body, so the body of the let is icoded as an expression body rather than as a command body, and the continuation inode and pattern are passed along to the icoding of the body so that the final value may be returned to the continuation. As we saw in the

rangeproduct example, icoding nested *let* expressions creates a chain of continuations. New continuations are added before the current continuation if the nested *let* is in the arms, and after the current continuation if the nested let is in the body.

7.5.3.4 Race expressions

Race expressions are icoded very much like parallel *let* expressions. The only differences are that a race continuation serves as the joining continuation rather than a joining continuation created from a *let* body, and the parallel subexpressions of the race are all icoded with the same pattern since the race continuation only has one reply handler — for :value. Otherwise the result looks much like shown on page 92 in figure 6-20.

7.5.3.5 Simple expressions and the continuation pulling optimization

Occasionally there are situations where a simple (non-continued) expression may need to be icoded. For example, if an *if* expression contains a simple expression in its *then* body but a continued expression in its *else* body, then the *if* expression must be continued. However, if it takes the *then* branch, there is no ask expression to make the bridge to the next continuation. We saw a situation like this in our rangeproduct example, and this situation also exists in the simple recursive factorial: in the base case a simple value can be returned. but the inductive case a recursive ask needs to be made. There are three approaches to dealing with this situation:

- One strategy is to create a bridge to the continuation by making a request to an *identity* actor which simply replies with the value(s) of its message parameter(s) to its customer.
- Another strategy is to simply send a reply message with the value directly to the continuation.
- A third strategy is to eliminate the message passing entirely and perform the body of the continuation using the value that was to be returned. This can be done by substituting the body of the continuation into the current context.

The third strategy, sometimes called *continuation pulling*, is the most desirable since it eliminates the communication entirely. However it has a few drawbacks: pulling the continuation body means that code is duplicated, and it doesn't duplicate other capabilities which are associated specifically with that continuation. The code duplication problem is not serious; as we have seen, the primitive behaviors produced are quite short, and code sharing begins again with the bridge to the following continuation. However, joining continuations also synchronize concurrent replies and store the returned values, as well as continuing the computation once the replies have been received. This functionality cannot be duplicated by simply duplicating the body of the joining continuation, so this approach cannot be applied when the continuation is a joining continuation.

The first and second approaches can be applied in this situation; since these approaches do send a reply to the joining customer, it can perform its synchronization duties and update its state. The current

compiler uses the first strategy when the customer is a joining customer because it preserves the structure of the transactions for the debugger. The current debugger (appendix B) records message passing patterns in terms of transactions: requests and their corresponding reply. Sending a request directly to the customer would produce a reply to which there was no corresponding request. Preserving this pattern is helpful for the programmer, since it produces a trace where all the transactions associated with a particular script are displayed at the same calling depth (indentation) (see appendix B for more details about Traveler). If we use the customer pulling strategy whenever the continuation is not a joining continuation, then simple expressions can be icoded as follows.

```
icode-expression(simple-expression current-body continuation pattern)
  (if (single-continuation? continuation)
    (then (let ((cont-body (continuation-inode-body continuation)))
            (let ((binding (make-binding (pattern-identifier pattern)
                                         simple-expression))
              ;; Bind the value in a let and perform the continuation body
              (let ((let-inode (make-let-inode binding cont-body)))
                (add-command let-inode current-body)))))
    (else (let ((request-inode
                  (make-request-inode (lookup 'reply-identity)
                      :do simple-expression
                      (body-sponsor current-body)
                      continuation (pattern-keyword pattern))))
            ;; Create bridge to continuation with request to reply-identity
            (add-command request-inode current-body)))))
```

We have seen an example of continuation pulling in our rangeproduct example (figure 7-6, page 103). The first continuation (cont1-behavior) may just return the value of lo, so the behavior of cont8-behavior, which is merely to reply with the value to the customer, has been pulled into the *then* arm of the *if*. The value of lo is first bound to the identifier expected by the body of the continuation, v0, and a duplicate of the body of the continuation can be found in the body of the *let*.

7.5.4 Exception Handling

Exception handling specified with a *let-except* indicates what to do if the expression in the arm cannot be evaluated because of a complaint. Therefore, not only are the exception handlers added to the continuation which processes the body of the *let-except*, but also to continuations generated as a result of icoding the arm expressions. In this section we look at one method by which the icoding process can add exception handlers to continuations, and the special characteristics of the this method. As I've mentioned before, exception handling is still at a primitive stage in Acore, but it may be instructive to see how one complaint handling mechanism is implemented.

We have seen that a *let* generates a continuation behavior which carries out the body of the let. In section 6.6 we saw that exception handlers are translated into complaint handlers of the continuations generated by expressions in the scope of the complaint handler. One way to implement complaint handling is to add the complaint handlers specified in the *let-except* to the continuation created, and

icode the handlers. Since this continuation is passed to the arm expressions as they are icoded, any further continuations generated can simply copy the complaint handlers from the continuation. If there are nested *let-except* statements, the new complaint handler may be merged with the old ones, omitting the handlers which are overridden by new versions.

One exception to the merging strategy concerns concurrent arm continuations. Recall that in section 6.6 we concluded that concurrent arm continuations should forward complaints to the joining continuation rather than processing complaints themselves (unless the arm has an exception handler of its own). Therefore, in the concurrent arm continuations the default complaint handler should forward complaints to the joining continuation rather than copying complaint handlers from the joining continuation. Just as values were forwarded through reply-identity, this is performed by forwarding complaints through complaint-identity.

One good characteristic of this method is that even though each continuation behavior has a complaint handler, if the complaint handler specifies further transactions which require continuations, the complaint handler continuations are shared between the behaviors. If we were instead to copy the exception handling into each *let* at the source level, these would be duplicated for every *let*.

An important characteristic of this method is that only expressions which are necessary for returning a value to the *let-except* are covered by its exception handling. Expressions whose values are ignored or are used by commands or expressions not directly involved in computing the result are not covered by the exception handling. This is usually the best solution, since the exception handler indicates what to do if the expression is not successfully computed, so the expressions independent of and concurrent with the computing of the value of the expression should not invoke that exception handling. For example:

```
(let-except ((money-order (let ((money (:withdraw my-account amount)))
                                (:print money debugging-stream)
                                (:create money-order :amount money)))
             (order-form (:new-order order-forms :item desired-item)))
            (except-when
             ((some-error (&rest ignore))
              "Sorry, I can't make the order now -- try again later."))
  (:send-order company order-form money-order))
```

The :print expression and the :create expression are evaluated concurrently, and the value of :create expression is returned to the joining continuation which has the complaint handler. The :print expression is there just for effect; its value is ignored. Since the value of the :print is ignored, it doesn't have direct influence on the success of the transaction, and since the requisition may be sent off and the receipt returned before the :print generates a complaint, the complaint handler shouldn't be invoked. The complaint handler only specifies what the *let-except* form should return if the body cannot be performed because an arm wasn't evaluated successfully. The only problem with this solution is that the :print statement may *appear* to be covered by exception handler.

7.6 Collecting Identifiers

The final step performed during the icode pass is to collect the sets of identifiers referenced in each continuation behavior. This is done after each form is icoded, and each inode transfers the sets to its parent inode. This has been left out of each of the procedures loosely sketched above for brevity.

To collect all the identifiers, the entire inode tree must be traversed. Since simple (non-continued) command and expression pnodes are inserted into the inode graph directly, *script* expressions hidden within these forms may be found. If so, they must be processed by the behavior separation phase and the following identifier propagation phase, since the simple commands and expressions won't be traversed again until the final Pract generation phase. This explains the recursive call to the icode phase in figure 7-1 (on page 96).

(It may be possible to partially collect the identifiers during the parsing phase and put them in places where they can be retrieved for identifier collection without traversing all simple commands and expressions. If this were the case, then *script* expressions would have to be separated into behaviors and their acquaintances propagated as soon as they were parsed, since their free identifiers need to be known before identifiers are collected in the expression enclosing the script. However, identifiers shouldn't be collected beyond the simple expressions during the parsing phase since the icode phase introduces new code during optimizations, and identifiers from the new code need to be collected as well.)

Five sets are collected. The free-externs are the pnodes of the identifiers which parsed as free identifiers; they will become the free identifiers of each behavior. The compiler-externs are the set of free identifiers added by the compiler; for example, the compiler generates calls to the reply-identity actor, so the identifier 'reply-identity' needs to be added as a free identifier to the behavior and is collected as a compiler extern. The compiler externs are kept separated from the normal externs because they are also propagated to the leading script by the next phase. The reason for this is that all free identifiers inside a script expression must be known before the identifiers referenced by the script expression can be known for collection outside the script. Most of the free identifiers of the script expression were collected during the parsing phase; recall the discussion about storing identifiers in the environment at the script boundary in section 7.4.5. The free identifiers added by the compiler must be propagated and added to this set, but there is no reason to go through the overhead of propagating all the free identifiers, so the compiler externs are kept separately.

The third set collected is the set of free local identifiers (free-locals). This is the set of identifier pnodes referenced which were not parsed as free identifiers for the whole script (this is where typing the identifier pnodes comes in handy), but are not bound in the current primitive behavior. This is the set to the left of the vertical bar in figures 7-8 and 7-9 (pages 105 and 106).

The fourth set is the set of defined local identifiers (defined-locals). This is the set of identifier pnodes for those identifiers which are defined within the current primitive behavior, by being bound

either in a primitive *let*, by a message handler, or, in the leading script, by being an acquaintance. This is the set to the right of the vertical bar in figures 7-8 and 7-9.

The fifth and final set collected is the set of continuations referenced. This set is collected so that the following phase which traverses the directed acyclic graph of primitive behaviors can do so without searching through all the commands for the other behaviors referenced. The continuation behaviors will eventually be referenced as identifiers which must be declared as free identifiers for each primitive behavior.

7.7 Walking the DAG

The result of the separating phase is a directed acyclic graph (DAG) of leading and continuation script inodes, each filled with a tree of commands and simple expressions. As we saw from the graphs of figure 7-9 (page 106), one of the jobs of the next stage is to propagate the local identifiers so that each continuation has references to the actors it needs. While walking through the DAG, two other jobs which need to be done are performed as well: linearly ordering the primitive scripts, and collecting the compiler externs for the leading script. Each of these jobs is simple by itself and independent of the others, so they have been combined into one pass.

As we saw in section 6.7 the definition of all the scripts must be an expression. A series of nested *let* expressions serves our purpose, but the compiler must find an linear ordering of the scripts by which each primitive script is defined before it is used. Such an ordering exists because the graph of scripts is acyclic; there are no loops in Acore behaviors, so there are no cycles of references between the primitive behaviors. One way to build this ordering, which is used by the current compiler, is to make a depth first traversal of all the paths through the graph, adding each primitive script to the front of a list as it is encountered advancing. The idea is that each time an edge is crossed, it is coming from a script which references the continuation behavior reached, and since the continuation behavior reached must be defined before the script, it is added to the front of the list to guarantee this. You can think of the process as reordering the list of primitive behaviors, moving the encountered behavior to the front of the list at each step. However, since memory is available, we save having to traverse the list searching for and deleting the element from the middle of the list each time by just adding a new element to the front of the list, and after we're done, traversing the list once deleting duplicates. Once the list has been completed and the duplicates deleted, the list is stored with the leading script inode from which it will be recovered when it is time to generate the final Pract code.

In figure 7-9 we saw that the local identifiers referenced by a continuation need to be passed from the point where they were bound, through the acquaintances of the continuations between, down to the continuation. To propagate the identifiers from the points where they are referenced back up the DAG to the nodes where they are defined, a depth first traversal is required so that all the references in the behaviors below a node in the graph are propagated through that node as needed. However, once all the

references below a node are propagated up to the node, its acquaintance set is the set of all references which need to be propagated to that node and any nodes below it. Therefore there is no reason to explore the graph below the node again; thus the cost of this propagation is proportional to the number of edges in the DAG (plus a factor depending on the number of a identifiers propagated). The cost of just traversing the graph for this tree is the same as finding a spanning tree, proportional to the number of edges. Since only a depth first traversal of all the edges of the DAG is necessary for this job, combining it with the complete traversal of the previous job just requires checking at each node whether the node has been previously encountered.

One minor point: Note that the DAG of references propagated between behaviors is not the same as the DAG of message passing bridges between behaviors. While the forks are the same, the joins are not the same. Forks occur when either a decision is made or expressions are evaluated concurrently; in order for the fork to occur, multiple messages must be sent and existing references must be passed to multiple continuation behaviors. Joins due to decisions may occur in both graphs; the flow of control may pass through the decision, split through either arm of an *if* expression, and join again once the value of the *if* expression is computed. This effect produces the fork and join in figure 7-9. However, joins due to a parallel *let* do not occur in the reference propagation DAG. The reason is that the joining continuation is created at the fork, not by the concurrent arm continuations, so no references are passed by acquaintances from the concurrent arms to the joining continuation. Thus, in the graphs of figure 7-9, the parallel *let* arm continuation behavior, represented by the node farthest to the right, is not connected to the joining continuation behavior below it. Instead, the joining continuation behavior is connected to the behavior containing the corresponding fork, since the joining continuation must be created beforehand so that it is known to all concurrent transactions. However, the last continuation behavior at the bottom may be instantiated by two paths depending on the decision taken in cont3-behavior.

In the previous section we saw why compiler externs are collected; this stage collects the compiler externs from all the primitive scripts and stores them in the leading script inode. Collecting the compiler externs only requires visiting each node once and adding any compiler externs defined there to a set. This is easily combined with propagating local identifiers, since both check for redundant visits to nodes.

Thus, the traversing DAG phase is simple, but important. It is potentially time consuming on complex DAG's, but so far it hasn't been a problem.

7.8 Connecting Behaviors

The result of traversing the behavior DAG is that all references are propagated through the acquaintances of the behaviors from the point where they are bound to the continuations where they are needed. The next step is to connect the behaviors with the commands for creating or updating continuations. There are five things to be done to accomplish this:

- Deciding whether to *create* or *update*.

- Positioning acquaintances in the acquaintance lists.
- For creates, deciding how many additional ignored slots are needed.
- For updates, deciding which acquaintances need updating.
- Deciding where to insert the *update* or *create* for the next continuation.

7.8.1 Deciding whether to *create* or *update*

Each leading script handler and each continuation body is a tree of commands. Each body may have multiple commands, and bodies within *let* or *if* commands may also have multiple commands. Multiple commands from a body of concurrent Acore commands or a parallel *let* form concurrent forks; decision points in *if*s form forks to exclusive paths. At the leaves of this tree are the *request* command inodes which are connected to the following continuations; we've seen this in all the primitive behaviors. The challenge is to go through this tree of commands, knowing which of these requests must create a new continuation, and which may reuse the current continuation, and add the commands for doing so.

The possible situations can be itemized as follows:

- The current body is in the leading behavior. This is always the initial case in any tree of behaviors; as we saw in section 6.2, the leading actor should not be used as a continuation. Therefore, in the leading behavior, all continuations must be created.
- The current body is in a continuation behavior, and there is a parallel branch where several continuations will be needed concurrently. In this case, only one of the branches may may reuse the current continuation. Not all the branches may need a continuation, so the method for choosing which branch is allowed to reuse the current continuation must take this into account.
- The current body is on a branch which isn't allowed to reuse the current continuation. This is like the initial case in the leading behavior: all continuations must be created.
- The current command is an *if* and the continuation may be reused. An *if* has two bodies which are mutually exclusive; only one of them will be performed. Therefore, if the *if* may reuse the continuation, both of the bodies are permitted to reuse the continuation.
- The current body is the dispatching node of a parallel *let*, and the continuation is available to be reused. In this case both the joining continuation and any concurrent arm continuations are possible candidates for reusing the current continuation; this is like the case of parallel commands in the continuation body. One one of the connections may be an update; the rest must be created.

One possible method of making the decision which satisfies all these situations is to make use of information at each fork indicating which of the subcommands will need continuations. This information is available in the form of the sets of referenced continuations collected at each inode, which I described at the end of section 7.6. Thus at each concurrent fork, these sets could be consulted, and one of the forks which needs a continuation would be passed a flag allowing it to reuse the continuation; the rest would be passed a flag indicating a continuation must be created.

However, if a flag must be passed around anyway, a simpler method of making the decision is to

pass a mutable flag around, and the first branch to use the continuation sets the flag, indicating it is no longer available for the other branches. This eliminates the need for decision making at each concurrent fork, at the expense of some overhead at each exclusive fork (*if*) — even though one branch of an *if* may set the flag, the other branch may still be able to use it.

Different strategies may be used for deciding which of several concurrent commands may reuse the current continuation actor. One possibility is to compare the continuations to see which will reuse the most state, in the hopes this will improve efficiency by reducing the average size of the created continuations and the amount of work the update must do. Another is to do some sort of analysis to see which is likely to survive the longest, if this helps the memory management system by keeping the other continuations as short-lived as possible. The current compiler takes a relatively naive approach, in the interests of keeping compile time down. It uses the following simple ideas: since the value of an expression body is determined by the last expression in the body, the last expression is most likely to need to reuse the most state, since it needs at least the customer and reply keyword. Since it determines the value of the expression, it may also be of most interest to the programmer; keeping the most interesting transactions in one continuation actor allows the programmer to see more from the lifeline of that actor (see appendix B for more on lifelines). Therefore, the last commands in a body get first crack at reusing the continuation. The other simple idea is that joining customers are likely to reuse more state than the arm continuations (for much the same reasons), so the joining customer is made by updating the current continuation rather than creating a new actor whenever possible.

7.8.2 Positioning Acquaintances

Once the decision is made whether to update or create a continuation, the next step is to find what positions the next behavior expects its acquaintances. As I discussed in section 7.2.8, the acquaintance positions can usually be optimized so that updates only add new acquaintances without rearranging existing ones. The current compiler doesn't make an extraordinary effort to optimize the acquaintance positions considering multiple routes through the behavior DAG, but it does make some optimizations which can easily be accommodated in one pass. Continuations which are created keep their original acquaintance positions, which may be whatever resulted from identifier collecting. To order the acquaintances of a behavior which is updated from a previous continuation behavior, first match the positions of the acquaintances the two behaviors have in common, then fill in any holes and add to the end any new acquaintances of the later behavior. If there are fewer acquaintances in the later behavior, then there may be some ignored positions which may or may not be used by subsequent behaviors.

There are three issues to keep in mind while implementing this algorithm. First, there may be multiple paths through the DAG to the behavior, so once the ordering has been set by one connection it should not be changed. This means that in a behavior connection DAG with many joins, it may be desirable to do all the updates before the creates in order to take advantage of existing ordering as much as possible. The current compiler does not make this effort.

Second, as I discussed in section 7.2.8, later behaviors may need more acquaintances than the first behavior of a continuation actor, so it may need to be created with some ignored acquaintance positions. Computing the maximum size needed by any one continuation actor requires knowing the graph of possible behaviors to which it may be updated. Since the decision as to which connections are updates and which are creates is not made until this stage, this maximum must be computed while traversing the graph during this stage. This can be accomplished by running this pass in a depth first manner, propagating the ordering constraints downward from the root and propagating the maximum size needed back upward toward the root.

Third, the set of acquaintances needed by a behavior is not always just the set of free locals which it needs, but in the case of a joining continuation also includes the storage for a unique value and the incoming values. A good example of this was discussed in section 7.2.8.

7.8.3 Building a *create* expression

Once the decision has been made that the continuation must be created, the positions of the acquaintances of its first behavior are known, and the maximum size of the behaviors it may update to is known, then it is a simple matter to build the *create* expression. The identifier for the script is stored in the continuation inode. The values for the acquaintances will be bound to the same names in the current environment, either as acquaintances of the current behavior or because they are defined (first bound) in the current behavior. The end of the *create* may be padded with 'ignores.

(*create behavior-identifier {acquaintance-names...} {*'ignore *padding...}*)

There are two examples of *creates* in our RangeProduct example in figure 7-10 (page 111), one in the leading script and one in the fork for the parallel *let* (cont5-behavior).

7.8.4 Building an *update* command

Once the decision has been made to update the current continuation actor with the next behavior, and the positions of the current continuation behavior and next behavior are known, it is a simple matter to build the *update* expression. The script, acquaintance 0, is always updated; the identifier for the new script can be retrieved from the next continuation inode. To build the rest of the *update*, the acquaintance lists of the two behaviors are compared, padding with 'ignore where one is shorter than the other. Any positions which are different must be updated.

(*update* self (0 *behavior-identifier*) *{(diff-acq-position diff-acq-name)}...*)

If the new behavior is a joining continuation, then the acquaintance positions which will be used to hold incoming values must be initialized to the unique value used to tell whether or not the incoming value has been received. The code which creates the unique value must be built as well, so the *update* is wrapped in a *let*.

```
(let ((unique-value (create null-script)))
  (update self (: behavior-identifier) {(diff-acq-position diff-acq-name)}...
        {(incoming-acq-position unique-value)}...))
```

Because the compiler takes the care to create the original continuation actor large enough for any behavior it may take on, there is no need to check whether the behavior is large enough at this point. Note that a position is updated to 'ignore when the following behavior doesn't use it. Although this seems like unnecessary work, it frees up references to actors which are no longer needed, which may allow them to be garbage collected or may give the memory manager a hint that the previously referenced actor is no longer needed here and may be migrated somewhere more useful (see appendix A for a brief discussion of migration).

(One note about the examples: For clarity I have been using self as the identifier to refer to the current continuation actor in the primitive behaviors. As we can see from the examples, there needs to be an identifier referring to the current continuation so that it may be reused as the customer in requests as well as being updated. However, the identifier self is also the default identifier used at the Acore level for referring to leading actor. Therefore, to avoid this conflict of names, the Acore compiler actually uses the identifier current-continuation in Pract continuation scripts instead of self.)

7.8.5 Deciding where to insert the *update* or *create*

Once the *create* expression or the *update* command is built, inserting it is straightforward. You may recall that the separation (icode) phase left holes for the continuation, since at that stage is is unknown how the next continuation will be connected. These holes are represented by <??> in figure 7-6 (page 103). These holes are found in the customer positions of many *requests*, and in the binding of the joining customer in the dispatching inode at the fork. If the continuation is created, then the *create* expression built is simply inserted into the hole. If the continuation is updated, then the identifier self bound to the current continuation is inserted into the hole, and the *update* command built is inserted close by, either in the same body as the *request* or, in the case of the joining continuation, in the body of the *let* which binds the identifier used for the joining continuation. This can be seen by comparing figure 7-6 with figure 7-10 (on pages 103 and 111).

7.9 Generating Pract

After the update commands and create expressions are inserted, the Pract code can be generated fairly straightforwardly. The only code which isn't a direct translation of its inode structure are the script lists, single continuation behaviors, and joining continuation behaviors. Rather than repeat a large example here, the points expressed below can be elucidated by examining the forms of the scripts in figure 7-10 on page 111 (except for nesting *lets* to bind scripts, omitted for clarity).

The script list built in the traverse-DAG stage is stored in the leading script inode. In the final list,

the leading script must be at the end of the list, since it is added first and no continuation refers to it so it is never added to the front. At the head of the list is one of the continuations at the leaves. Thus the list is in the same order as the series of nested *let*s must be, so a simple recursive algorithm builds the script list described in section 6.7. Each continuation script is given a name when it is created; this name is used to refer to the script by code which uses it, so this name is now used for binding the scripts in the *let*.

Single continuation inodes are translated directly into Pract *script* expressions. The acquaintance list comes from the acquaintance list built and ordered in the DAG-traversing and behavior connecting stages. The list of external identifiers comes from the set of free identifiers classified by the parse phase and collected during identifier collection, the set of compiler externs from identifiers in commands added by the compiler, and the set of names of continuation behaviors referenced by this continuation. A single continuation has one reply handler, whose pattern is determined by identifier used in the *let*, and whose body is body of commands to be performed by the continuation. Complaint handlers which were added now become part of the script as well. All the continuation behaviors in figure 7-10 except cont7-behavior are examples of continuation behaviors expecting a single reply.

Joining continuation inodes are translated into Pract *script* expressions in a similar manner. The only difference is that a reply handler is generated for each arm of the parallel *let*, and a test expression to decide whether to store the value received or proceed with the body of the continuation is generated. The test expression tests whether all the other incoming values have been bound or whether a transaction has been aborted. For example if three values a, b, and c are expected, then the tests may look as follows:

```
(script
 ((reply-keyword customer sponsor c b a unique-value) (...)
  (is-reply (:c c)
    (if (or (== unique-value ':abort) (== b unique-value) (== a unique-value))
     (then (update self (4 c)))
     (else <do body...>)))
  (is-reply (:b b)
    (if (or (== unique-value ':abort) (== c unique-value) (== a unique-value))
     (then (update self (5 b)))
     (else <do body...>)))
  (is-reply ((:a a) :self self)
    (if (or (== unique-value ':abort) (== c unique-value) (== b unique-value))
     (then (update self (6 a)))
     (else <do body...>)))
  (is-complaint (some-error ignore &rest args)
    (if (== unique-value ':abort)
     (then (update self))
     (else (update self (7 ':abort))
         (complain-to* customer
           (some-error reply-keyword args))))))
 ...)
```

In figure 7-10, cont7-behavior is also an example of a joining continuation behavior.

7.10 Summary

In this chapter we have taken a look at how an Acore compiler can translate Acore code into primitive behavior as Pract code. In three passes at increasing levels of detail, we have looked at the different stages in the compilation of Acore: macroexpansion, parsing, separating into primitive behaviors, propagating acquaintances, connecting behaviors, and finally generating Pract.

This compiler differs from most other compilers for two primary reasons: the source code describes a concurrent rather than sequential language, and the output code is in terms of primitive Actor behaviors rather than random access or stack oriented machine. These factors mean the compiler must deal with trees and directed acyclic graphs of control flow rather than a single thread, and must explicitly propagate the references needed through this graph rather than depending on an auxiliary run time environment or stack. Some of the methods may be similar to those used by highly optimizing compilers to consider rearrangement of code and allocate registers, but here they are used purely to deal with concurrency in the actor model, which provides a simple, clean basis for thinking about concurrency. Since this compiler compiles Acore into primitive actor behaviors, it has not been concerned with optimizations but simply information flow and control flow through concurrent systems of actors.

The contribution of this implementation is to show how the actor model can be used to implement a concurrent language. While the Pract code presented here compiles into actors and Lisp functions without much trouble for our current emulator (see Appendix A for a few more details), additional work may need to be done to compile series of behaviors into an efficient form for a message passing parallel computer.

In the following chapter, I will look into other languages and the models on which they are based and compare them to Acore and actors.

Part IV
Comparisons

Chapter Eight

Comparisons with other Concurrent Languages

We have described the design of Acore and seen how it is based in the Actor model of computation, both in concept and in implementation. We now have a background for comparing it with the designs of several major concurrent programming languages. These languages differ in the nature of concurrent processes which may be easily expressed. I will examine how well each of them meets the goals described in Chapter 3, especially how well they express simple cooperative and competitive processes. I will also focus on several aspects of their design which affect their concurrent nature: how concurrency is expressed, how concurrent processes communicate, and how state change may be shared between concurrent processes.

How concurrency is expressed. A language may encourage or discourage the use of concurrency by the ease with which concurrent processes may be expressed, and this affects how people think about using concurrency. For example, in Chapter 3 we noted that it was desirable to be able to express cooperative and competitive concurrent processes. If simple forms of cooperative concurrency can be expressed concisely, then divide and conquer algorithms can be expressed in their concurrent form without losing any clarity. On the other hand, if competitive concurrency is difficult to express, then people may not even think of using it in their programs.

How concurrent processes communicate. Processes may communicate both directly through messages or streams, and indirectly through effects on shared objects. The ease by which concurrent processes communicate affects how people will organize processes.

How state change may be shared. There are many organizations which are best modeled in a concurrent environment by systems of objects which change state. Less obviously, objects which change state are needed for representing changes in the availability of a shared but limited resource over time. They are also useful for organizing shared information. For example, many concurrent algorithms can take advantage of dynamic programming to share partial results between concurrent processes; this is facilitated with a shared database.

In this chapter I will consider four classes of concurrent languages. These classes cover the most important work in concurrent programming languages which has come to my attention: languages based on communicating sequential processes, languages based on concurrent calls to sequential procedures, functional and dataflow languages, and concurrent logic languages.

8.1 Languages based on Communicating Sequential Processes

One approach to designing a concurrent language is to take the sequential process model familiar from multitasking operating systems and and incorporate it in a language with a system for communication between the processes. This is the approach taken by Hoare's Communicating Sequential Processes (CSP) [Hoare 78], its commercial offshoot, Occam [Pountain 85], and by Ada [US DoD 83]. These languages can be characterized as follows:

- *Expressing Concurrency*: Concurrent processes are created in these languages by commands or declarations which spawn new tasks. For example, in Occam the statements nested within a PAR represent concurrent tasks; in Ada tasks are specified by a **task** declaration.
- *Communication*: Concurrent processes communicate in these languages using commands for sending and receiving a synchronized assignment through a communication channel. These commands are like assignment commands in that they bind a variable at the receiving end. The communication channel specifies the protocol accepted for the communication. Since communication is synchronized, both the sender and the receiver must be ready before communication takes place; this eliminates the need for hidden buffering.
- *Sharing changes in state*: Since the primary mechanism for synchronizing processes in these languages is to rendezvous for communication, the primary mechanism for sharing state changes is to share access to a process which encapsulates the state. (Ada also allows shared variables, but rendezvous still serve as synchronization points.)

The primary problem with this approach is that it is *verbose*. To use a concurrent process, separate statements are required to create a new process, to communicate with a process, and then to use a value thus communicated. This is apart from the declaration needed to specify the communication protocol. The overhead of communicating with a process is alleviated somewhat by the fact that a process inherits the values of free variables from its defining context, so a new process can easily be initialized with many parameters; however, any results computed by the process must be returned over a communication channel.

Another problem is that the command orientation of these languages does not lend itself to describing the communication structure clearly. For example, division of labor is very common use of concurrency. Yet the purpose of the joining process which collects the results may not be apparent without studying its code, so it may not be obvious whether concurrency is being used in cooperative or competitive ways, or even which of the other processes are producing the results. As a result of this problem and the verbosity mentioned above, using concurrency for simple division of labor, e.g. for divide and conquer algorithms, pays a price in lost clarity.[5]

Control of processing is associated with the processes in these languages. Each process can be given a static priority; in Ada a process may also be aborted. This approach is a start, but because the priorities are static, it is missing the flexibility of Acore's sponsor mechanism to redirect attention based

[5]CSP and Occam don't allow recursion (although there are bounded work-arounds using replicated processes), so they aren't particularly suitable for recursive divide and conquer algorithms. But division of labor into parallel subprocesses is still common.

on the situation at hand.

While not a direct consequence of this approach, these particular languages were designed with different goals than the goals we developed for Acore, so it isn't surprising that they fail to meet several of the goals we outlined in chapter 3. For example, none of these languages are object oriented[6], none have garbage collection, and Ada tasks are the only construct remotely like higher order closures.

More interesting to look at are the differences in the underlying approaches to concurrency. For example, the model used in CSP and Occam envisions processes as physical objects connected by communication lines. Thus, processes are statically allocated and statically organized. Since communication lines are point to point, communication between these objects is synchronous and unbuffered. The Ada model isn't as static, but it also assumes synchronous communication.

In contrast, the actor model which supports Acore assumes the actors are dynamic, active objects. Since new actors are frequently being created, the population of actors is constantly changing. Uneven population growth results in migration to equalize the distribution of the actors. Since acquainted actors may become distantly separated, rather than force actors to wait during communication delays, an asynchronous communication model based on a mail system is used. We feel this model is a closer match to the object (and pointer/record) oriented processes performed in largely symbolic computation.

The contrast in these underlying approaches has a few architectural implications. Supporters of the actor model expect the fine grain division of computation into actors with local state will facilitate load balancing between processors in a highly parallel computer. In contrast, the larger amounts of state associated with sequential processes are harder to move around, especially if part of the state is shared with other processes. Of course, this is a tradeoff. Processing a message for a primitive actor on an actor machine is a very small task, so the instructions/message ratio is relatively small. This means that task switching is high, so current multitasking architectures with large process states and long switching times may not be suitable for running actor programs. On the other hand, programs of the sequential process model with greater instructions/message ratios can take better advantage of those architectures at the expense of less concurrency and less flexibility.

8.2 Concurrent Calls to Sequential Procedures

One of the primary objections to the communicating sequential process approach is the loss of clarity due to communication through channels and to verbosity resulting from the command oriented approach. Therefore, one approach to fix this problem is to introduce concurrency in an expression oriented manner within an expression oriented language. This is the approach taken by Multilisp [Halstead 85], a dialect of Lisp for multiprocessors.

---

[6]Although it seems possible to use Ada tasks as expensive objects.

- *Expressing Concurrency*: Concurrent processes are created in Multilisp through pcall and future expressions. Expressions without these two forms are evaluated sequentially as in other dialects of Lisp. A pcall expression is similar to normal function call evaluation in Acore: new tasks are spawned to concurrently evaluate the parameters to pcall, and after they have all returned values, the function call is performed. A future expression is similar to the *future* expression of Acore: it creates a *future* object which can be passed as the value of the future expression and spawns a new (sequential) task to concurrently evaluate the expression determining its real value. Tasks which attempt to access the future before the value has been evaluated are suspended until it has been determined.
- *Communication*: The primary means of communication between concurrent processes is through sharing the environment where they were created and by returning a value when they terminate, either direction to the waiting process which performed a pcall, or to the future representing the value.
- *Sharing changes in state*: Multilisp tasks may share state change through side effects on the shared environment or on shared objects. There is no mechanism forcing the safe performance of any mutations, but primitives (semaphores and replace-if-eq) are provided for building abstractions which can provide atomic mutation.

Simple cooperative concurrency due to division of labor is easily expressed using pcall and future. The expression oriented syntax is much more concise than the command oriented syntax of the languages of the previous section. In one simple expression, concurrency is spawned, values are returned, and the values can be used as parameters to an enclosing expression. The communication pattern is also much clearer from the nesting of the expressions.

Although no expression oriented syntax for describing concurrent competitive processes is included in the original Multilisp language, it shouldn't be too difficult to add one. For example, a form similar to the *race* expression in Acore could easily be introduced, provided it is possible to decouple the future from the task which computes its value. Thus the value of a future may be partially determined by the completion order of the tasks. A parallel cond can also be introduced.

However, flexible control of processing may be more difficult to add. Rate of execution may be controlled to some extent by giving tasks (processes) priorities. However, a mechanism similar to the sponsor mechanism of Acore may be needed in order to provide the same ability to adjust the priorities of running processes as circumstances change. Acore sponsors can be arranged in a hierarchy of sponsors, with subsponsors for subprocesses, permitting control over subprocesses through a parent sponsor. Similarly, in Multilisp, a hierarchy of sponsors could control the priorities of the tasks and subtasks in their care. This auxiliary mechanism is easily modeled using actors in Acore; perhaps a different metaphor can be developed which is more suitable for Multilisp.

The expression oriented syntax for expressing concurrency makes .. very easy to create short-lived processes which terminate once they have returned a value. Thus, in Multilisp, processes are not used to encapsulate state as in CSP style languages, but only to perform some action. Therefore most processes need communicate only by inheriting an environment at initialization and returning a value upon termination. Communication between initialization and termination is needed only when processes share

partial results; this sharing is accomplished through shared environments and data structures.

Multilisp and Acore are similar on the surface in many respects, modulo the differences pointed out above, due to their common heritage from the Lisp community. However, there are two underlying differences which separate the approaches take by the two languages.

The first difference is that Acore encapsulates all state changes inside actors, while Multilisp depends on the programmer to make sure any changes which can be seen by concurrent processes are performed atomically. This has two consequences. First, this means programs are less secure in MultiLisp, since inconsistent changes may result if a concurrent program doesn't mutate shared state atomically. This kind of bug is very difficult to find since it depends on the relative timing of the tasks performing the mutation; yet this timing is nondeterministic and may not be repeatable. While it may be simple to check that this problem doesn't arise while the program is being written, especially if functions are small and don't mutate the environment often, there is still the possibility that an error will be introduced later when maintaining or trying to optimize the code by adding more concurrency with an additional future or pcall. The second consequence is that since environments are mutable, they must be shared between concurrent processes. If the processes are performed in parallel on separate processors, there is an expense associated with maintaining consistency, either from remotely referencing the environment or from ensuring the local copy is consistent. In contrast, bindings in Acore are immutable, so expressions may be evaluated concurrently on separate processors by copying the bindings without worrying about consistency.

The designers of Multilisp are aware of this problem, but, as far as I know, haven't proposed a solution. Two possibilities come to mind: One is to take the CSP/Occam approach and disallow mutations to environments not local to the current process. If this is too restrictive but we would still like to avoid the overhead of maintaining consistency between concurrent processes, it may be possible to determine at compile time which bindings are needed by a subprocess and are not mutated (most of them), and duplicate these bindings if the process is migrated.

The second underlying difference is that Multilisp is based on a sequential process model with its associated stack and environments, while Acore is based on the message passing actor model. In implementation, this means that Multilisp can take advantage of current processor technology to run coarse grain processes. However, this advantage depends upon the locality of the state needed by the process. The sequential process is associated with a processor, so it runs best when the state needed by the process is local to the processor and few remote memory accesses need to be made. But after load balancing, large data structures may be distributed across many processors, so processes which traverse large data structures (e.g. graphs, knowledge bases) may find many memory accesses are remote accesses, and may spend time waiting for communication delays. The large task state then becomes a liability, slowing down task swapping time and making it expensive to migrate tasks to balance the load better.

In contrast, the actor model implementation does not associate activation with a process and processor; instead messages are sent to actors and processed by the processor on which the actor resides. After load balancing distributes large structures of actors across many processors, memory accesses are still local, and interprocessor communication consists primarily of higher level (and hopefully sparser) requests and replies rather than low level memory requests.[7]

This second difference represents a tradeoff. The Multilisp approach is optimized for the situation where a task primarily accesses state local to the task, perhaps generating new structures. Tasks are expensive to migrate, so they are migrated primarily to take advantage of concurrency rather than locality. However, as multiprocessors become larger and more processors are added, more structure is distributed across more processors, so the likelihood that the state needed is local to the current processor drops. Thus, the actor approach is optimized for the situation where the state needed for a process is distributed across many processors, so to take advantage of locality, control for the process hops around to processors with messages. It is hoped that with new processor designs which take advantage of VLSI, message driven processors can reduce the latency of message response so they can also be competitive when locality is strong.

8.3 Functional and Dataflow languages

In the previous section we saw that languages based on concurrent calls to sequential procedures improved the clarity of programs by introducing an expression oriented syntax for expressing concurrency. Functional and Dataflow languages such as Id Nouveau [Nikhil et al 86] have taken this to the extreme, specifying that all independent expressions can be evaluated concurrently. They are based on the lambda calculus, and have encouraged its use for describing *functions* by preserving determinacy.

- *Expressing Concurrency*: Even more than in Acore, concurrency is the default in functional languages; it is expressed whenever expressions are independent. Since expressions are referentially transparent, a variety of evaluation strategies are possible, including normal order (delayed) evaluation, as well as the applicative order evaluation used in Acore.
- *Communication*: In functional languages, the only means of communication between concurrent processes through the free variables in expressions and the values the expressions return. Id Nouveau extends this with *I-structures* whose slots behave as futures: they may be written only once, and they queue requests for reads which arrive before a value is written.
- *Sharing mutable objects*: Representing shared mutable objects is outside the scope of functional languages. Functional languages strive to be determinate by preserving the Church-Rosser property, so expressions can be reduced to a unique normal form no matter what order the reductions are performed. If it were possible for subexpressions to mutate and read a shared mutable object, then different results could result depending on what order the subexpressions are reduced. However, there is some research on going to introduce *managers* into dataflow languages. Managers would represent shared mutable objects, e.g.

[7]Some memory requests for Multilisp must, like messages, specify operations to be performed remotely — for example, atomic test and set operations such as replace-if-eq.

for managing limited shared resources. One representation of the state of a manager uses a private stream; processing a message from the input stream and the current from the private stream state produces a result value and the next elements for the next state in the private stream. This model is similar to replacing behaviors in actor model.

Since functional languages are based on the lambda calculus, concurrency due to division of labor for divide and conquer algorithms is easy to express. Subexpressions are concurrent, so recursive calls within a lambda expression may be performed concurrently.

Because functional languages and extensions to them such as Id Nouveau are determinate, it is impossible to express competitive processes which rely on arrival order nondeterminism. Since all processes must complete, there is no mechanism for explicitly controlling the relative resources allocated to concurrent processes.[8] The addition of managers introduces the possibility to make arrival order significant, so competitive concurrency may then be possible.

Dynamic programming to share partial results between processes is one form of sharing state changes which can be determinate. Since the state changes are all monotonic (adding new information to the tableau), it is just a matter of arranging for some process to add them. In many situations this process can be described statically in the algorithm. Alternatively, through the use of delayed futures, the first process which needs a particular value in the tableau may trigger its computation. However, more general control structures for managing the shared database require being able to test whether the value has been (or is being) computed yet, but no method has been found for making such a test without introducing constructs which destroy the Church-Rosser property. Thus, without managers, changes of state can be shared only in limited ways.

This approach is close to the actor approach in several ways, but since the underlying goals are different it isn't surprising that it fails to meet some of the goals of Acore. The most important difference is the stress on determinacy in these languages, which contrasts with Acore where indeterminacy is needed to express competitive concurrency.

In implementation, dataflow architectures and actor architectures have a similar flavor, one which makes no use of sequential processes. Instead, messages are routed to processors which process them using only local information. However, there is a small difference in what people think of as the stereotypical computation for each architecture. In actors, since the stereotypical application is symbolic processing of some arbitrary sort, the stereotypical situation is performing an ask expression (similar to a function application). Thus the values from evaluating several subexpressions are received and synchronized by a joining continuation, which then sends off the request to perform the ask expression. In some cases this can be optimized so the continuation performs the operation itself, such as creating a new actor (e.g. a cons cell) with the parameters. In dataflow, the stereotypical situation is evaluating an

[8]This is not to be confused with the work for optimizing resource use among processes behind the scenes, e.g. with loop control.

expression involving primitive arithmetic, constructive, or selective operators such as +, cons, car, etc. In this case, the primitive parameters are conceptually sent to copies of the operators. The processor performs the synchronization if there are two parameters, and then the operation is performed on the parameters. In some cases a function must be applied, so in this case the arguments are synchronized and collected into a chain, and the function is applied. Thus, dataflow architectures emphasize optimizations for concurrent evaluation of primitive expressions, whereas actor architectures emphasize controlling higher level messages and keeping track of the many actors created. Due to the similarities, people working on dataflow architectures and actor architectures will be able to learn something from each others' experiences.

8.4 Concurrent Logic Languages

Now for a rather different approach. Concurrent logic languages such as Guarded Horn Clauses (GHC) [Ueda 86] and Concurrent Prolog (CP) [Shapiro 83] grew out of efforts to speed up Prolog by taking advantage of concurrency between clauses and subgoals. However, in pursuing concurrency, the languages were modified to the point where they now express a rather different semantics than Prolog. Programs in concurrent logic languages are not performed by backtracking interpreter; instead, clauses should be read procedurally. For example, GHC can be characterized as follows:

- *Expressing Concurrency*: A predicate is defined by several clauses of the form:

  $H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n$.

  The guard of the clause consists of the clause head H and the guard goals $G_1, \ldots, G_m$ — i.e. everything before the commitment operator '|'. The guards for all the clauses of the predicate are tried concurrently, and the predicate (nondeterministically) commits to the clause which succeeds first. The clause head H is a pattern which is unified with the incoming goal. If it matches, then the clause attempts to satisfy the guard goals with the resulting bindings. Once the predicate has committed to a clause, the other clauses are stifled and it attempts to satisfy the body goals. Thus concurrency is expressed in concurrently matching the clause patterns, in concurrently trying to satisfy the guard goals concurrently, and if it commits, in concurrently trying to satisfy the body goals.

- *Communication*: As in Prolog, the primary means of communication is through unification of variables in patterns with terms. Unification provides a write-once mechanism for synchronization, similar to futures and I-structures. However, unification is not permitted to instantiate variables for the guard goals, and only permitted to instantiate variables for the body goals once the predicate has committed to that clause. This prevents any backtracking to undo bindings.

- *Shared Mutable Objects*: Mutable objects can be represented in concurrent logic languages through the use of streams [Kahn et al 86]. Successive states of the object are represented as successive elements of a state stream, and successive messages to the object are represented as successive elements of an input stream. Communication is performed between objects by unifying the message with the tail of the stream, thus defining the next message in the stream. Since several concurrent processes can't unify messages into same tail of the stream, each message sender must have its own stream to the receiver, and these input streams must be nondeterministically merged.

From the procedural reading of a predicate's clauses, we see that simple cooperative concurrency

is easily expressed as AND-concurrency between the guard goals or the body goals. Thus subexpressions (subgoals) are concurrent in these languages as well.

Competitive concurrency arises in the form of competitively evaluating the guards of several clauses but committing only to one of the clauses. This mechanism can be used to build the nondeterministic merge of two streams, so a concurrent competition which produces a stream of values in the order they were discovered may be expressed. However, stifling these competing processes is the only control of processing expressed in the language. Since no variables may be instantiated to satisfy guards, competing processes which may be stifled may not perform any instantiations which can be seen after the clause is stifled. This avoids the problems of aborting transactions and reverting state which arise in Acore when stifling computations, but the cost of this convenience is to throw away any possibility of keeping the intermediate results which may have been obtained. Thus, matching clauses is most useful as a clause selection mechanism read procedurally rather than as a mechanism which allows stifling processes. Competitive concurrency is more generally expressed by communications through nondeterministically merged streams, without any control of processing.

Languages based on concurrent logic languages may be developed which better suit our purposes. For example, preprocessors such as Vulcan [Kahn et al 86] greatly improve the utility of the language for object oriented purposes, and show that concurrent logic languages can be as powerful as actor languages such as Acore.

Many differences between logical objects in Vulcan and actors in Acore are ones which are primarily relevant to personal preferences in programming languages. For example, the logical style of Vulcan can be concise for simple programs, especially when destructuring patterns are used. However, the flat logical style can be less clear than an expression oriented style. For example, programs expressed as nested expressions in an expression oriented language must be expressed as a flat set of subgoals in logic oriented languages. The communication between these subgoals is specified by variables, which is less clear than the nesting of expressions — it may not even be clear which subgoal determines the value of a variable and which subgoals just read the variable.

The problem of flat structure of logic programs also shows up in the lack of modularity or higher order behaviors. Modularity is desirable for partitioning large programs so name conflicts aren't a problem. However, in logic programs, since there is no lexical scoping of names, all predicates must appear at top level. This problem can be fixed by a preprocessor which translates a language with some encapsulation into the flat logical form by uniquely renaming clauses as they are globalized. Lack of higher order behaviors seems to be a more difficult problem, but it is possible that it may also be handled with a suitably designed preprocessor.

Thus, we find that through a series of transformations the concurrent logic model seems to provide many of the same capabilities as the actor model. However, the actor model is a closer match to the organization of systems as objects and thus provides a more direct framework for implementation.

8.5 Summary

Each of the languages we've considered in this chapter has some ideas to offer, but none satisfy all the goals we set out to meet in Chapter 3. Most of these languages — Multilisp, Id Nouveau, and the concurrent logic languages — are research languages still under development, and may yet develop in ways bring them closer to meeting our goals. For example, task control and competitive concurrency may be added to Multilisp, managers are under investigation for Id Nouveau, and preprocessors which allow higher order behaviors to be expressed naturally in concurrent logic languages may appear. However, none of the languages provided any form of control over processing which is nearly as flexible as the sponsorship mechanism of Acore. It will be interesting to see if descendants of each of these languages finds a niche in the future marketplace for concurrent languages.

Part V
Conclusions

Chapter Nine

Summary and Future Work

In this thesis we started out with the premise that new programming languages are needed to open up the full possibilities for experimenting with concurrent processing. We've taken a look at some of the overall design goals we'd like to see such a language meet. It should have the generality to express diverse kinds of concurrent processes, easily expressing simple forms of cooperative and competitive processes as well as control over processes. One of our conclusions was that a first step would be to construct a language from which further experimentation in concurrent language design can develop. Thus it should be built from a small set of powerful abstraction features, combining higher order abstraction, as found in function oriented languages, with the message passing abstraction, as found in object oriented languages, and a powerful macro facility, all in a uniform manner. We created a uniform vision by basing the design on a few central ideas from the actor model of computation and the lambda calculus. The most important of these ideas are the model of a concurrent process as an organization of communicating actors; the expression oriented syntax for describing behaviors, with lexical scoping of identifiers; and the replacement behavior model of state change. The result of working out a design from all these goals has been the design of Acore.

Since the conception of Acore has been based on the Actor model of computation, behaviors in the language are easily compiled into behaviors of primitive actors using the idea of continuations. Since these primitive actors process messages using only their local state, they are well suited to be distributed across a parallel computer architecture which can take advantage of the concurrency by processing messages for many actors in parallel. Such architectures can also take advantage of the small amount of state associated with each actor, which facilitates migrating actors between machines for load balancing. In the second part of this thesis, we saw how Acore behaviors can be implemented in terms of behaviors of primitive actors, and the work involved in the Acore compiler to make this transformation.

With the background of the Acore design and how it can be implemented in terms of primitive actors, we then looked at several other concurrent programming languages. Each of them met some of our goals, but since they were designed with different goals in mind, none of them quite fit our needs for experimenting with concurrent processes. Each of them has made a different set of tradeoffs, in the underlying conception as well as in the surface syntax, which makes it more suitable for one type of application than another. We hope that Acore will prove to be a sound foundation for experimenting with symbolic processing distributed over a massively concurrent computer.

What conclusions can one draw from this work? At the least, I hope the reader comes away from reading this thesis with the idea that there is some promise in the approach taken here. Before this promise can come to light however, many more issues which have surfaced during this development

need to be addressed:

- *Stifling and Complaint Handling*: The biggest issue revolves around the idea of stifling computations, and the complaint handling necessary to make sure this doesn't cause problems which cannot be handled by the system. Stifling is most useful for exploratory types of processes, such as for aborting branches of a parallel search or to stop the examination of the consequences of hypotheses no longer needed. As long as there is no need to undo any changes made by such exploratory processes — e.g. if no mutations are made, or if all working structure is thrown away, or all mutations are monotonic additions of information — then stifling isn't a problem. However, there is always the possibility that such a process may need access to a limited resource or that discoveries made by the process should result in changes to a shared data base. Either stifling must be disallowed in such cases, or they must provide complaint handling to revert things back into an acceptable state. Mechanisms for concisely specifying complaint handling for aborted computations need to be developed.
- *Exception Handling in General*: In order deal with problems in interacting with other systems or even problems between subsystems, programs will need to deal with failures and exceptions. In these cases, the decisions about how to proceed may depend on context rather than local information, so mechanisms for dealing with exceptions in a more general fashion than just aborting are needed. Also, the syntax of exception handling needs to be extended to apply more easily to expressions inside commands as well.
- *Sponsorship*: Acore is one of the first languages to provide a mechanism for controlling concurrent processes dynamically, so cliches for the use of sponsorship haven't yet developed. As experience grows, we may get a better idea of the common ways in which sponsorship is used, and perhaps will find new ways of incorporating it into concurrent languages to make these easy to express.
- *Modularizing Behavior*: Acore is an object oriented language, but unlike most object oriented languages it doesn't provide any way of combining or building on modules of behavior for defining new behaviors. We felt that no existing system is clearly superior, and there are new problems in adapting such systems to a concurrent environment. Thus, rather than commit ourselves to any one system, we left it open as an avenue of research which may be pursued using Acore, much as existing proposals are developed on existing languages such as Lisp. However, some mechanism needs to be developed soon, since complex objects and systems of related objects are difficult to understand and maintain without the modularity such a mechanism provides.
- *Hiding Forwarding*: Forwarding actors arise often in Acore, and their existence can be detected in some situations. This is fine, but we need to allow alternative forms of identity checks which specifically follow forwarding so that forwarding actors can be used invisibly if desired.
- *Typing*: Currently there is no default typing mechanism for actors. Acore *script* expressions do allow some type information to be supplied to the script, but by default this is left empty.

These are all important areas of development within the language. At this time, however, the most important work yet to be done is to develop a parallel computer system which can run actor programs efficiently. Only then will these ideas come to fruition. I hope that this work will help people interested in parallel architectures better understand the flavor of actor programs.

Appendices

Appendix A

The Apiary Emulator

In this thesis I concentrate on how one may express concurrent programs in Acore and how Acore programs can be compiled into primitive actors. I'm sure this may arouse some reader's curiosity about how a parallel computer might run programs of primitive actors, so in this appendix we look into the ideas behind the tightly networked architecture we call the Apiary and a few of the lessons learned in implementing a simple emulator for running actor programs on one processor.

A.1 Apiary Concepts

The *Apiary* is the name for our project to examine the issues and problems in creating a massively parallel computer for running actor programs. The name comes from the idea that, much as apiaries are boxes filled with busy bees, such a computer would be a box filled with busy processors, sometimes called *workers*. The architecture of this computer should be scalable, so additional processing power can be added for larger problems, just as additional memory is added to sequential computer systems today (figure A-1).

Each worker (processor) has its own memory to hold its queues and any actors which are residing in its memory. The actors in the system are distributed among many processors; each processor executes messages received for the actors which reside on the processor, and sends messages to other actors on other processors over the network. The tuple consisting of a message together with the target actor to which it is addressed is called a *task*. If a worker has many more tasks than its neighbors or is running low on memory, a load balancing mechanism *migrates* actors and tasks to less busy or less full workers. Since actors may be migrated from processor to processor, the *mail system* must keep track of the location of each actor so that messages may be forwarded to it. The mail system uses unique identifiers (*UID*'s) to identify each actor (figure A-2).

Each worker basically goes through an endless cycle of pulling a message off of its task queue, delivering the message to its target actor (which is a primitive actor), performing the handler, and putting any new messages created back on the queue. If messages are received over the network, then they may be added to the queue; the target of a message is not local to this worker, then the message will be sent out over the network.

Before this idea can work, many questions will have to be answered and techniques for coping with the complex management problems developed. Some of the unanswered problems are mentioned briefly below.

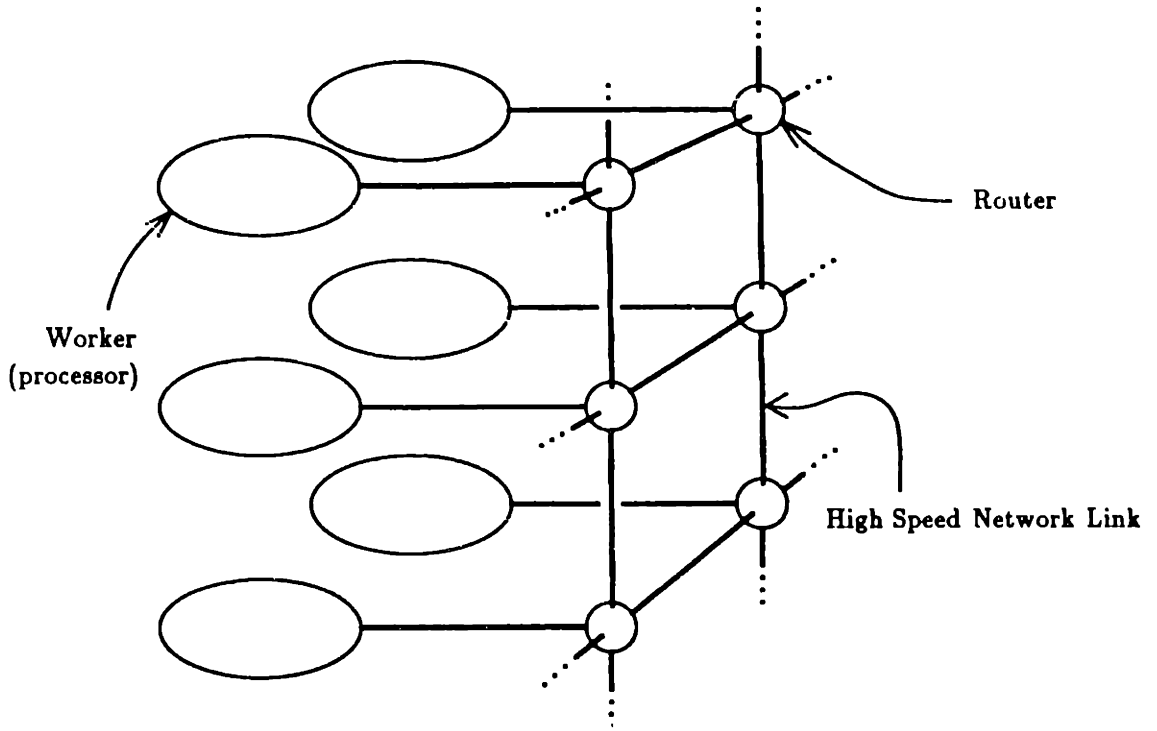
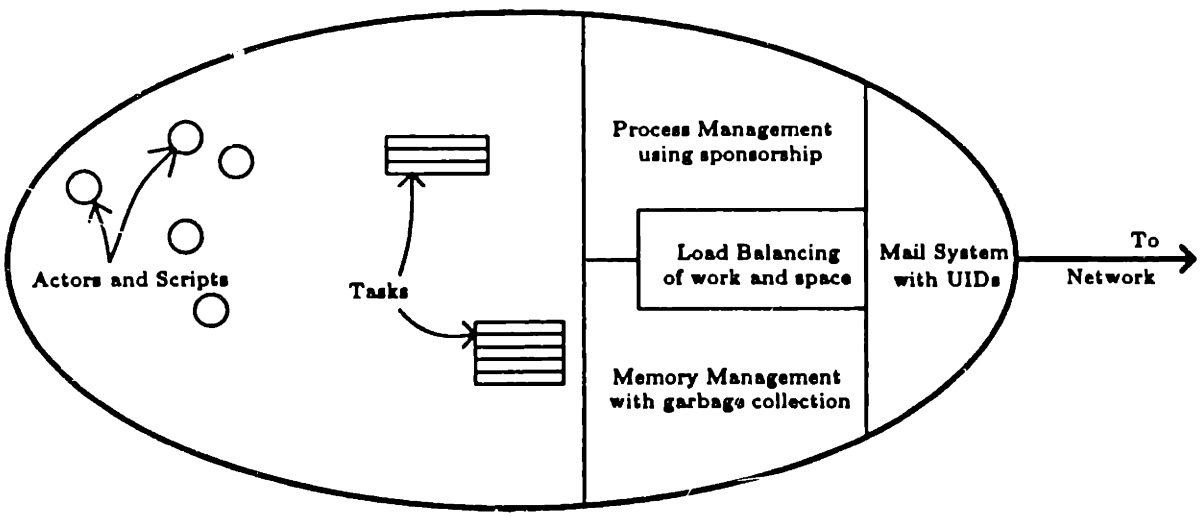


Figure A-1: Generic Apiary Network



- *The mail system.* References to actors in the form of UID's are spread through out the system. The mail system needs to keep track of the location of the actor which corresponds to each UID so that messages sent to the actor can be routed to the correct processor.

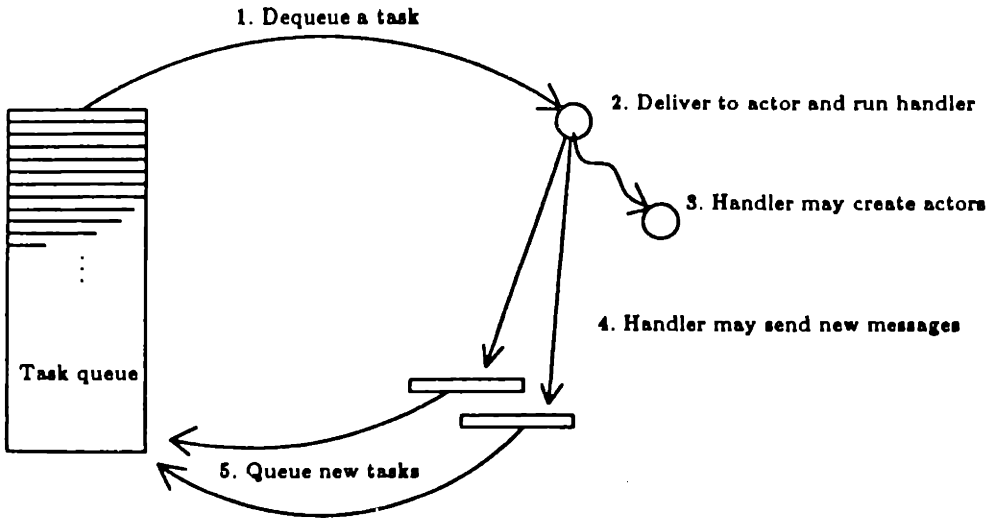


Figure A-3: A worker's work cycle

Reference tree schemes have been proposed [Halstead 79], but they are intended for low speed networks where processors do the routing, so other techniques relying on caching are being investigated.

- *Migration.* When actors move from processor to processor, the mail system must be able to keep track of the move. Messages sent to the old location must be forwarded to the new location; eventually the mail system should be able adjust so it can send the message directly to the new location.

- *Garbage collection.* Not only must the system keep track of where actors are, but it must also be able to reclaim storage taken by actors which may no longer receive any messages, i.e. which are not referenced by any other actors or in any tasks. The challenge here is to find a distributed garbage collection algorithm. The reference tree scheme gives one solution to this problem, but like all schemes similar to reference counting, it incurs an overhead in both processor time and memory.

- *Load balancing.* When some processors end up with more work and/or less memory than other processors, it may be time to balance the load between processors by migrating actors. Ideally we would like to be able to balance the load to take advantage of parallelism by distributing actors who receive messages concurrently to separate processors, while at the same time not introducing too many communication delays between actors who frequently send messages to each other. The challenge here is to find some way of determining which actors are the hot spots and which actors communicate so that load balancing isn't a blind guess. There are some similarities between this problem and page swapping in virtual memory, but here there is added complication due to multiple processors working on different problems or parts of a problem simultaneously.

- *Process Management.* Processors must manage processing of transactions within the resources controlled by the transaction's sponsors. However, as Arvind and Culler point out in [Arvind & Culler 86], processors may need to do much more than this in order to run concurrent programs within resource limits of the machine. The problem is that running transactions in parallel takes more resources (e.g. memory) than running them sequentially, since temporary storage for each of the transactions must be allocated simultaneously. If concurrency is eagerly exploited, the processor may quickly run out of resources, even if it

would have had much more than enough if it were to run the transactions sequentially.

A.2 The Emulator

The Apiary Emulator is a facility for running compiled Pract programs on a Lisp machine, although most of it is written in Common Lisp to make it easier to port to other machines. At this stage it has be developed primarily for running and testing programs, so my effort has been put into making it run at a rate fast enough to test out small applications. It currently emulates only one worker using one Lisp machine; I hope to extend it to emulate multiple workers on one Lisp machine and/or multiple workers on multiple Lisp machines communicated over a network (initially Ethernet). The ideas learned from these experiences will help us develop an operating system for running actor programs on a parallel computer.

The following sections outline the representation of actors in the Apiary Emulator and describe how the emulator runs actor programs, including sponsorship control.

A.3 The Representation of Actors

Primitive actors are represented in the Apiary Emulator as shown in figure A-4.

- The *mailbox* represents the actor locally; all local pointers to the the actor point to the mailbox. If the actor is not local to this machine (e.g. if it has been migrated to another worker, or just a reference to the actor was sent to this worker), then the mailbox will **not** contain the state, but only the UID of the actor and perhaps a hint about where it can be found. If the actor is resident on this worker (which is always the case for the current one worker emulator), then its *current state* will be present.
- The *current state* of the actor is a vector containing its current script and current acquaintances. This vector is separate from the mailbox because it is not needed if the actor isn't local, and its length may change as the actor changes behavior.
- The *incoming queue* holds messages for the actor which arrive when while the actor is locked (i.e. insensitive) processing a serialized handler. It also serves to indicate when the actor is insensitive.
- The *UID* is the unique identifier used to communicate references to this actor to other workers.
- The *biography* is a place to keep a history of debugging information about the actor. See Appendix B for more details about Traveler and the transaction recording mechanisms.

The mailbox of an actor corresponds loosely to the *mailing address* of the actor theory [Agha 86]. It provides a level of indirection for the state of an actor so that it may change its behavior over time. In our emulator, we also take advantage of this indirection to allow the actor to move from worker to worker. Since an actor has at most one mailbox on each worker, it may be migrated from worker to worker by moving its state vector from the mailbox on one worker to the mailbox on another, and updating the vacated mailbox with forwarding information. Local pointers to the mailbox do not need to

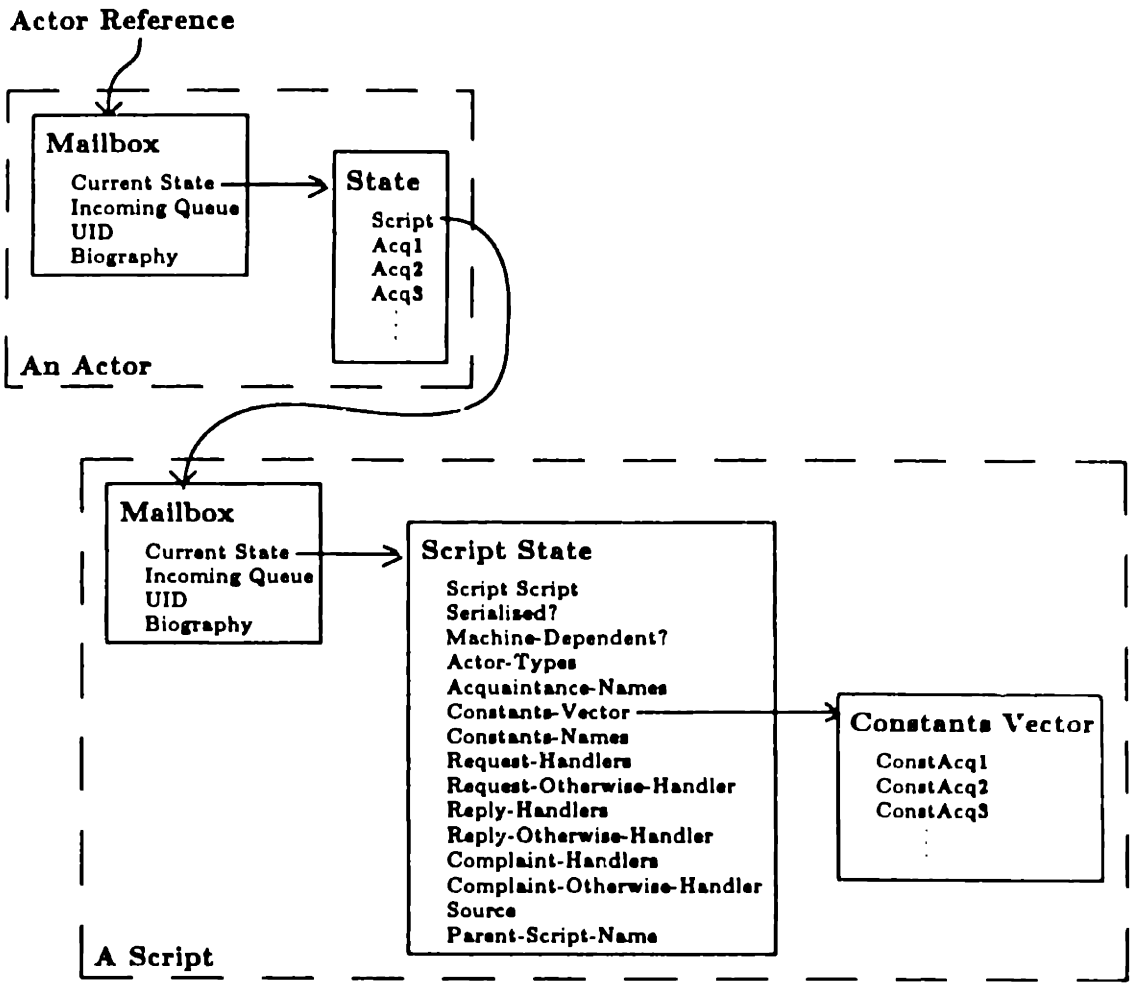


Figure A-4: Representation of actors in the emulator

be changed as the actor is migrated.

Although we have not yet experimented with this approach in a multiple worker setting, it appears to be the most efficient approach on stock hardware. The cost of this approach is that a little more memory is consumed on every worker with a reference to the actor, even if the actor isn't local to the worker. Actor machines may be able to circumvent this overhead by providing direct hardware assisted translation between UID's and processor nodes or UID's and the local address, so that a compact UID is a sufficiently efficient reference for both local references and remote references. A global memory space, as found in a shared memory machine, can provide cheap UID's at the expense of making it difficult to move an actor; this approach may be sufficient for small multiprocessors, but the inability to move actors easily makes it difficult to perform load balancing and optimize the performance of the machine.

A.4 The Representation of Scripts

In the current emulator, scripts are also represented as actors with the same structure (figure A-4). Scripts must have the acquaintances pictured to hold shared information about all actors with that script. This information includes the tables of handlers and the vector of free references (constant acquaintances) needed by the actor. The constants are kept in a vector rather than compiled directly into the handlers so that when the script is migrated, all the references to other actors from the script can be located and translated on the new worker, just as the acquaintances of an actor are. Serialized? is a boolean value which indicates whether any of the handlers of the script are serialized; if not, actors with this script will never change state and may be copied on several workers if they become a bottleneck. Machine-dependent? is another boolean value which, if true, indicates that the actor represents an interface to a device or is otherwise machine dependent and should not be migrated. The rest of the acquaintances contain debugging information; for example, the acquaintance names are used when the actor is displayed by Traveler.

A.5 The Representation of Handlers

The *handlers* of a script are represented as Lisp functions. The parameters to this function are the elements of the task. Thus, if the task is represented as a list, then to run a handler the emulator could just apply the function representing the handler to the task. For example, a request task may be represented in a queue as the following list:

(request *target customer reply-keyword sponsor*
recording selector message-parameters. . .)

Target is the actor to whom the message is being delivered. *Customer* is the actor to whom the reply to this request should be sent, with *reply-keyword* to select the correct reply handler. *Sponsor* is the sponsor of this transaction, the actor which should be consulted if ticks have run out, and the default sponsor for any further transactions started by the handler. *Recording* is a hook for the debugger to record tasks; see Appendix B for more details. *Selector* is the keyword which selects the message handler; it is included for otherwise handlers which may need to forward the message. *Message-parameters* are the actors which make up the rest of the message.

However, nearly every handler needs to get at the acquaintance (state) vector and the constants vector, so the emulator extracts this information from the target and its script ahead of time. Also, the symbol request serves to identify that this task is a request and this should be handled by a request handler; it is not needed within the handler. Thus, the actual parameters to the lisp function implementing a request handler are as follows:

(*acq-vector const-vector target customer reply-keyword sponsor*
recording selector message-parameters. . .)

Replies and complaints omit the customer, reply-keyword, and sponsor.

Handlers are Lisp functions with these parameters, where the parameters specified in the Pract

handler are used in place of *message-parameters...*, and the hidden parameters are given special names which won't conflict with names provided by the programmer. The body of the handler is the Lisp code which performs the handler; it returns no useful value.

Most of the constructs in Pract (see the *Pract Reference Manual* in Appendix D for descriptions), such as *if*, *let*, *or*, etc, are implemented directly as their Lisp counterparts.[9] Since we've designed the representation of actors so that each actor is represented by at most one mailbox on each worker, the == (identity) expression can be simply implemented as an eq. Special Pract expressions and commands such as *create* and *request* are implemented as calls to Lisp functions provided by the Apiary Emulator.

A.6 Builtin Actors

Builtin actors are the actors which are represented directly by the implementation, i.e. the actors which are not represented as we have described above. Although they are not strictly necessary, most implementations will take advantage of the representations of certain datatypes provided by the hardware. The Apiary Emulator is no exception: we take advantage of the Lisp machine's representation for booleans, numbers, symbols, and strings. Since these actors are immutable, they need no mailboxes and may be copied from machine to machine. Their scripts are known by the workers and derived from their type. The Lisp machine is a tagged memory architecture machine, so all objects have a type. To implement builtins on an untagged architecture, some representation which includes type information will have to be found, just as for other object oriented language such as Lisp or Smalltalk. The simplest thing to do is what we've done: implement it in Lisp and let the Lisp system take care of the types.

A.7 Sending Messages

Now that you know how actors, scripts, and handlers are represented in the emulator (figure A-4), and the basic cycle performed by a worker (figure A-3), it should be simple to see how to make this run. To start the worker, just put a task on its queue. Then:

1. The worker dequeues the task, and checks to see if target actor is local, residing in the memory of this worker.
2. If the target is not local, then look in the mailbox to see where the task should be forwarded, and send the task over the network to that worker.
3. If the target is local (always the case in a one worker emulator), then check if it is locked from processing a previous message.
4. If the target is locked, add the message to the target's incoming queue and find another task to run.
5. If the target is not locked, then extract the acquaintance vector from the mailbox, the script from the acquaintance vector, and constants vector from the script. Look up the handler in

[9] Well, *if* trades progn's for the *then* and *else*.

the script's handler table using the message selector, and apply the handler to task augmented with the two acquaintance vectors. (If it is an *update* or *replace* message, update the state of the actor and run the next message on its incoming queue, if any.)

6. As the handler runs, it makes calls to functions implementing commands like *request*, and *reply-to*. These are for sending more messages; as in figure A-3, these functions can add the new tasks to the queue. When the handler is done, the worker finds another task to run from step 1.

This simple loop is the heart of the worker, and it can keep cycling around until there are no more tasks to do, and then wait for more. Note that on a one worker emulator this loop may not emulate the nondeterminism of multiple workers if the task queue is operated as a strict FIFO queue. However, we can introduce nondeterminism to better emulate a parallel implementation by randomly choosing the next task to perform. All of the tasks on the queue are ready to be run, so if any one was delivered to another processor with no work to do, it could be run immediately. Thus, as long as our method of choosing the next task doesn't violate the guarantee of delivery (i.e. no task can be left unrun forever), then this nondeterminism is sufficient to emulate the nondeterminism of a truly parallel system.

Even in a multiprocessor Apiary, a sophisticated worker may take advantage of this flexibility to choose tasks for process management purposes. If other workers are idle, the worker may choose to run tasks which are likely to increase concurrency. If the system is becoming saturated, the worker may choose to concentrate on tasks which do not increase concurrency. For example, to increase concurrency, concurrent transactions may be performed in breadth first manner, while to discourage concurrency they may be performed in a largely depth first manner. As long as the guarantee of delivery is not violated, so other tasks in the queue are eventually run without too much delay, the worker has freedom to optimize choices for performance.

A.8 Optimizing Allocation

For many conventional processors, the above worker loop has one feature which can make it rather inefficient: it allocates memory to store every new task. To implement a worker on a conventional processor it is therefore a good idea to take advantage of the flexibility to choose the next task to reduce the memory allocation overhead. One way to do this is by *direct sending*: instead of putting the message on the queue, the message sending functions (e.g. request, reply-to, etc.) may deliver the next message sent directly to its target, in effect jumping directly from step 6 to step 2 in our simple worker loop when a message sending command is encountered. If the worker were to do this all the time, two problems would result: if it started running an infinite message sending loop, then it would stay in the loop, violating the guarantee of delivery; it may also eventually run out stack space. Therefore we put a limit on the number of tasks the worker may perform in this manner before it puts the remainder of the tasks on the queue and selects a new task from the queue. A convenient number is the worst case number of tasks which may be performed before stack space runs out.

Use of this technique has sped up the Apiary Emulator by roughly 20-30% or more on some

computations, running at most 50 messages before queueing. Setting the maximum higher has diminishing returns, since a relatively small percentage of tasks have large activation trees. (An activation tree is the tree of tasks, where the children of each task are the tasks it activated — the new tasks created as a result of running it — and the parent of each task is the task which activated it.)

Allocating continuation actors is also a source of overhead. The Acore compiler helps reduce this by updating rather than creating continuation actors when the current continuation can be reused. A good implementation should also make allocation and deallocation of continuations as efficient as possible.

At this point, someone who has just read through the comparisons with sequential process languages (e.g. Multilisp) in Chapter 8 may ask whether this mechanism doesn't show that sequential processes are more efficient, and whether the formulation of this structure as a sequential process wouldn't be more efficient, since calls might be made directly to functions or handlers rather than indirectly through the message sending functions.

The first thing to point out in response is that this is an optimization made for running actor programs on conventional processors. Since these processors are designed to run sequential programs, it is likely that they will perform faster on familiar ground. Actually, the primary advantage of using this technique is that it uses the stack to allocate and deallocate storage for the messages, so a non-sequential strategy which used some other efficient means of allocating and deallocating memory could be just as fast. This response also applies to the second half of the question: in a message passing architecture, the message passing functions would likely be replaced by message passing instructions, so the overhead of the function call is unnecessary.

The second thing to point out is that unlike sequential processes, messages are delivered in a depth first sequential manner *only as far as the target actors are local.* The worker doesn't get bogged down with remote memory accesses when the target isn't local; it just forwards the message and moves on to another. Therefore, in computations where the actors involved are distributed across the network, the actor approach may be superior for keeping processors busy doing productive work.

A.9 Sponsorship

We left out one point from our worker loop: how sponsorship fits into the picture. The reason for this is that we can take advantage of the optimization of the previous section to implement sponsorship reasonably efficiently. The basic idea of sponsorship is that the system must charge the sponsor one tick for every request. In our basic worker loop, we could make a request to the sponsor every time we pulled a request task off the queue (the request to the sponsor is sponsored by the system sponsor). If the sponsor returned the tick, we would be able to run the task. However, this is horribly inefficient, perhaps at least doubling the number of tasks run by the system.

But with our direct sending optimization, many tasks are run sequentially, exploring the activation

tree depth first. Most of the time subtransactions use the same sponsor. Therefore, the worker can spread the overhead of making a request to the sponsor over many tasks by making a request for a supply of ticks when a task is dequeued, and then running many transactions using that supply of ticks. During this run, a tick is charged from this supply for every request. If the supply of ticks runs out, then any remaining tasks must be queued. If any ticks are left over at the end, another request is made to return them to the sponsor.

Of course, if a subtransaction does happen to use a different sponsor, it shouldn't be run using the original sponsor's supply of ticks. In this case, the simplest thing to do is to put the request for the subtransaction on the task queue; when it is dequeued, a supply of ticks will be obtained from its sponsor.

A.10 Conclusion

The Apiary Emulator has proven to be a very useful tool for experimenting with Acore programs during the design of Acore and later for designing and testing its compiler. In its current state it runs only as a single worker on a single Lisp machine, but I hope to extend it to multiple worker and multiple Lisp machines to start getting into some real parallelism and to experiment with many of the issues discussed at the beginning of this chapter. It currently runs fast enough to experiment with small programs. The speed at which it runs is largely dependent upon the number of continuations which must be created; for example running simple programs such as recursive factorial of 100 and rangeproduct of 1 to 100, rangeproduct is twice as slow as recursive factorial.[10] Thus, additional work is needed to optimize the allocation of continuations if improvements in emulation speed are to be made.

The lessons to be learned from this experience are probably nothing that isn't obvious: make common operations as fast as possible. We have introduced a technique using the stack which may help allocation for emulators running on conventional processors.

[10]For an extremely rough ballpark comparison, recursive factorial of 100 in interpreted Lisp on the Lisp Machine is currently about 5 times faster than in Acore; recursive factorial in compiled lisp is about 3 times faster than interpreted. However, the stack size must be increased to run these computations in Lisp.

Appendix B

Traveler: The Apiary Observatory

B.1 Introduction

Observing and debugging concurrent actor programs on a distributed architecture such as the Apiary poses new problems not found in sequential systems. Since events are only partially ordered, the chronological order of events no longer corresponds to their causal ordering, so the execution trace of a computation must be more structured than a simple stream. Many events may execute concurrently, so a stepper must give the programmer control over the order in which events are stepped. Because of the arrival order nondeterminism of the actor model, different actors may have different views on the ordering of events. We conquer these problems by recording the *activation ordering*, the *transaction pairing*, and the *arrival ordering* of messages in the Apiary and displaying the resulting structures in Traveler's window oriented interface under user control, either after the fact in case of a trace, or while the structure is incrementally constructed in a stepping session.

Actor programs are concurrent at a fine grain, the level of message passing. Each actor can process its messages concurrently with other actors using only its local state; thus concurrency is engendered whenever an actor sends more than one other actor a message. The Apiary architecture takes advantage of this property by migrating actors between the different processors of the architecture to balance the work load across the processors. Each processor in the Apiary keeps a queue of messages which need to be processed by the actors resident in it. The conceptual cycle of the worker is to take a message off the queue, deliver it to an actor which processes the message, and send any new messages to their target actors. Sending the new messages involves either sending a message over the network to another processor if the target of the message is resident there, or just enqueueing it locally if the target actor is local. A debugging system for the Apiary must deal with the fine level of concurrency and the distributed nature of the machine.

Actor programs are written in the core actor language *Acore*. Acore syntax is much like that of Lisp, but expressions and commands in the same context (e.g. a command body or an argument list) may be performed concurrently. Acore forms are interpreted in a message passing style, so for example

```
(:+ 1 2)
```

sends the actor '1' the message ':+ 2', i.e. the message with selector ':+' and a single parameter '2'. Symbols in the keyword package (starting with ':') are interpreted as the selector of the message. Function calls are also interpreted as a short hand for message passing with the assumed selector ':do', so for example

```
(list 1 2) and (:do list 1 2)
```

mean the same thing: the target actor 'list' is sent a ':do' message with parameters '1 2'.

B.2 Observing Transactions

Traditionally, tracing a program involves setting up an output stream (say, the user's terminal) and printing a description of an event on the stream as it occurs. This is feasible because the sequential nature of the program means there is a single total ordering on events, and this ordering corresponds well to the causal ordering of events. In this ordering, any functions called by a procedure are called one at a time in sequence, and all the subroutines called by the function are completed before moving on to the next function. Thus each function call is completed before the next begins, and the source of each subroutine call is apparent. For example, see figure B-1.

```
Procedure1 a b c
 Function1 a
  subroutine1 a
  <-- a'
  subroutine2 a'
  <-- a''
 <-- a''
 Function2 b
  subroutine3 b
  <-- b'
 <-- b'
 Function3 a'' b' c
 <-- (a'' b' c)
<-- (a'' b' c)
```

Figure B-1: Traditional Trace

Since everything is sequential, showing events in chronological order produces a stream where causality is apparent; there is no question who called subroutine3 or where the parameter b came from.

However, tracing a concurrent actor program cannot proceed so simply because messages sent to different actors may be processed concurrently. If we were to naively output the events to a stream as they happen, then the stream would only reflect the approximate chronological order of the events rather than the causal order. Since the events do not necessarily have a total ordering, their order may change from trace to trace. In a distributed architecture, not only will the order of concurrent events be nondeterministic, but the order in which descriptions of events arrive at the output stream will also be nondeterministic. Thus we sought a method of recording the causal order of events which could be implemented in a distributed fashion.

Given an object oriented distributed computing environment, the obvious thing to do was to build an object oriented distributed recording mechanism. A message and its target comprises a task; each task execution is a message reception and is considered an event. For every event, a task record is created which records the target and the contents of the message, the task record of the event which activated it

(the activating event), and the task records of the events which it activates (the activated events). Thus the *causal* or *activation ordering* of a transaction is recorded in this doubly linked graph of task records. See figure B-2.

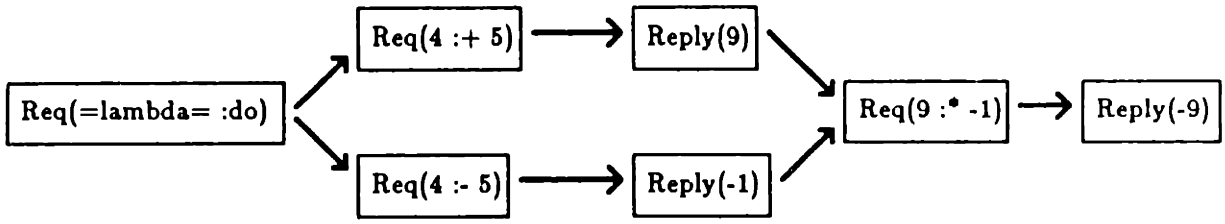


Figure B-2: Graph of task records

This is the graph of task records for the computation performed by the expression

((lambda () (:* (:+ 4 5) (:- 4 5)))

:Do is the

selector used for function calls.

In order to produce this graph while the program is running, each task in a transaction being recorded carries an extra parameter which holds information about recording. The Apiary emulator traps on the presence of information in this parameter and performs some special processing to record the task. If the parameter is empty, then the emulator does not trap, makes no recording, and runs at full speed.

A reference to the activating task record is passed using the recording parameter of the task. After a task is dequeued, but just before it is delivered to the target actor, a task record for the task is created. The passed task record is the activating task record for the new task record. A message is sent to the activating task record to link it with the new task record. Finally the message is delivered to the actor with the new task record in the recording parameter; this will be passed onto any new tasks the actor creates.[11]

Now that the causal ordering of the tasks in a transaction is recorded, the question arises about how to display it. Since the traditional trace displayed only a single sequence of events and we wanted to display a partially ordered graph of events, it seemed logical to try to find a graphical way of using the screen and somehow gain another dimension to the display. A *Tree display*, which showed the graph as a tree of event nodes connected by lines representing the causal links, was tried; a simple display looked

[11]This method does not quite produce the graph shown in figure B-2; in particular, of the tasks forming a join (e.g. the replies from the sum and difference), only the last to arrive will be linked as causing the continuation to continue (e.g. start the multiplication). The reason for this is that the replies are sent to a joining continuation actor, but the continuation actor just remembers the values previous to the last without activating any more tasks. Improving on this would require knowing how the joining continuation actor decided when to continue, an added complication we have been able to do well without so far.

somewhat like figure B-2 rotated 90° clockwise (except that joins were not displayed; see footnote). An *Actor Lifelines display*, which looked something like the event diagrams in [Clinger 81] and [Agha 86] rotated 90° counterclockwise, was also tried. These graphs were fine for showing the parallelism and structure in the computations, but they failed to be useful debugging tools because they obscured too much information about what the computation was doing and why. With all the nodes and lines, there wasn't much room to display much information about what actors were receiving messages, or what the content of the messages was. Another major problem was that these displays did not correspond to the abstractions made by the programmer in the program. The programmer thinks in terms of function and method calls and the results returned by these; in the event graphs there was little to link the requesting event with the corresponding reply, and the display wasn't suited to displaying it. In light of this experience, a display of the computation in terms of nested transactions, much like a traditional sequential trace, seems to be the answer.

Thus one goal was to create a display which looked something like a nested trace. For example, for the computation of figure B-2, it might look something like figure B-3.

```
Request(<lambda> :Do)
  Request(4 :- 5)
  Reply(-1)
  Request(4 :+ 5)
  Reply(9)
  Request(9 :* -1)
  [Tail Recursion]
Reply(-9)
```

Figure B-3: Design of the Transaction Display

A possible transaction display for the computation
`((lambda () (:* (:+ 4 5) (:- 4 5)))).`

There are several things to note about this format. The pairing of requests and replies makes it apparent which response corresponds to each request. The nesting of subtransactions preserves some of the causal information, but it does not show which of the subtransactions were performed concurrently and which must have been performed sequentially. However, we have found that the clues for debugging are generally found in the text of the descriptions (what is *that* actor doing here?, why did *that* function get called?) and in the nesting of transactions (what method was this called from?); causal relations between transactions within a method are much less important and are easily deduced from the source code (or they can be annotated on an individual task basis from the Traveler's display).

This display seems to work well, but more information is needed before it can be generated from the recorded graph of the transaction. The pairing of requests with the corresponding response is crucial to the display, but the graph does not directly contain information linking pairs of requests with

responses. A first attempt is to just indent a step for each request and outdent for a reply, but this approach doesn't work if there are tail recursive replies which close off more than one request.

Each request contains a *customer* to which the reply for the request should be sent. One way to make the pairing is to start with the request and search the activated graph for the reply which is sent to the customer of that request. However, such a search would consume large computational resources, especially in a large graph with many requests and replies, so a more efficient approach was needed. Thus we use *transaction customers* to put in a bit more effort to store the pairing information while recording.

The role of a transaction customer is to intercept the reply before it is sent to the real customer, record the information about the reply in a reply task record, link the reply task record with the request task record, and forward the reply to the real customer. Therefore, at the time a request task record is made (just before the request is delivered to the actor), a transaction customer is created with references to the request task record and the real customer, and substituted for the real customer in the request task. Thus the transaction customer masquerades as the real customer, and when it receives the reply, performs its duties and forwards the reply to the real customer.

With the transaction pairing links added by the transaction customers, the recording mechanism for transactions is complete, and generating a trace display is not difficult. A transaction display for Traveler is shown in figure B-4.

```
(#<SPONSOR-REQUESTER Actor~6966758> :More-Sponsor-Ticks APIARY::REQUEST #<<Tc
  (#<DEFAULT-TOP-LEVEL-SPONSOR-SCRIPT Actor~9543323> :More-Sponsor-Ticks 50)
  ←(:Value 50)
  (#<<TopLevelExpr> Actor~9543303> :Do)
    (#<RECURSIVE-FACTORIAL Loader-Forwarder~35667927> :Do 3)
      (#<RECURSIVE-FACTORIAL Actor~35667949> :Do 3)
        (3 :< 2)
        ←(:V6 NIL)
        (Update (Script: #<RECURSIVE-FACTORIAL Cont Script 35668028>))
        (3 :- 1)
        ←(:V4 2)
        (#<RECURSIVE-FACTORIAL Loader-Forwarder~35667927> :Do 2)
        ←(:V5 2)
        (3 :* 2)
        ←[Tail Recursion]
        (Update (Script: #<RECURSIVE-FACTORIAL Cont Script 35668054>))
      ←[Tail Recursion]
    ←[Tail Recursion]
  ←[Tail Recursion]
←(:Value 6)
```

Figure B-4: Transaction Snapshot

A snapshot of a Traveler screen showing the recursive factorial of 3. The transactions in italics have their subtransactions hidden.

The implementation of the transaction display for Traveler includes the ability to selectively open and close transactions to expose or hide subtransactions, giving the user control over the level of detail shown, much like outline processing in some modern word processors. Closed transactions (those whose subtransactions are hidden) are displayed in italics. Traveler also provides the ability to save a snapshot of the entire display in an editor buffer, so if the display is still too large to conveniently view on the screen even when irrelevant transactions are closed, it may be printed out.

The tracing facility provided by Traveler so far has proven to be very useful in localizing problems in small applications. Even when the program hangs, a trace of an incomplete transaction can be produced to find the reason for hanging. The ability to selectively open and inspect subtransactions has proven to be a great help in dealing with the complexity of small and medium small applications, but for larger applications it has become apparent that a form of selective tracing (e.g. displaying only messages to certain actors) is also needed to cut down even further on excess information.

B.3 Stepping Tasks and Transactions

Stepping in a sequential programming system is traditionally done either by modifying the compiled code of the program to be stepped to introduce breakpoints between statements, or by using an interpreted version of the program so it can be stepped with an interpreter. Both of these pose problems in a concurrent system where the program may be shared – modifying the program to introduce breakpoints or replacing the compiled version with the interpreted version also affects anything else which calls the program concurrently. Although an interpreted stepper can afford a source code oriented view of the stepping process, it cannot step compiled code. Yet it may be important to see how a compiled system actor interacts with other actors, so it would be nice if there was a way to step compiled programs without modifying them.

Actor programs on the Apiary already have a set of built-in breakpoints: each time a message is sent, the message may potentially need to be transmitted to another processor. Therefore there is a break each place a message is sent, and it becomes natural to step by messages. The acceptance of a message by an actor (the processing of a task) is an event; as the result of processing the message, new messages may be sent. Since message passing occurs at the level of function and method calls, message events have proven to be a small enough step size.

In order to step by event without modifying compiled code, we again make use of the recording parameter of tasks and the trap which checks it. To step a single task, an actor called a *collector* is placed in the recording parameter; the idea is that all new tasks created when the stepped task is performed will be sent to the collector instead of being delivered to their targets. When the recording trap finds a collector in the recording parameter, it sets a flag in the parameter before running the task. The recording parameter is passed on to all new tasks created by the actor; just before these tasks are delivered, they will also trigger the recording trap. This time, since the flag is set, the recording trap sends the tasks to

the collector instead of delivering the message to the target actor.

Thus, by using collectors, we can build a stepper which starts with a task, steps it by sending it with a collector, and then displays the new tasks returned to the collector. When another unstepped task is stepped, the cycle is repeated.

Concurrency in the program means that many tasks will be available for stepping simultaneously. If several of these concurrent transactions interact with the same history sensitive actors, it may be important to test different orders of execution. To control the order tasks are performed, the user must be able to select which task to step next from the pool of unstepped tasks.

Stepping concurrent programs also introduces a problem of presentation – how can the pool of unstepped tasks be displayed so that their relation to the operation of the program isn't lost? A traditional stepper displays the stepping process as an incomplete trace in progress. With a little effort we can take advantage of a dynamic window interface to display stepping concurrent programs as a concurrent trace in progress. The display dynamically expands as more activated transactions are filled in between the outer transactions.

In Traveler the same display is used displaying a trace as displaying the stepping progress, so once a transaction is completely stepped, the display which results is the same as if the program had simply been traced. See figure B-5. Since the same display is used, the facilities for opening and closing transactions are still available to control the display.

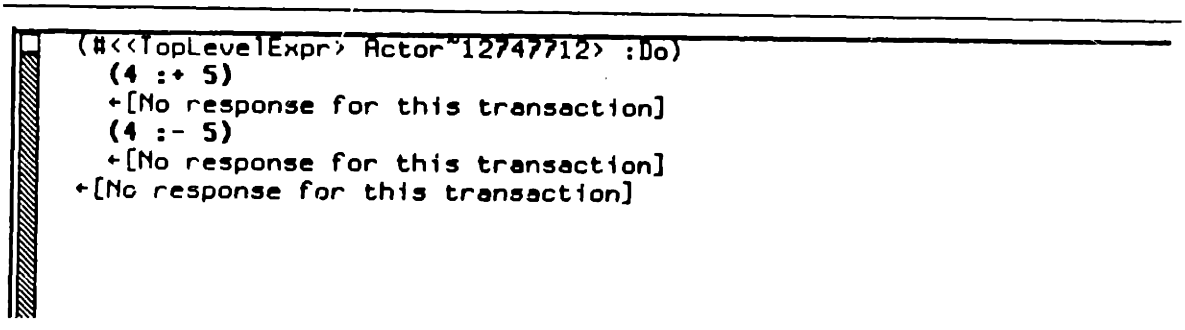


Figure B-5: Snapshot of the Stepper Display

This is a snapshot of the stepper in the middle of computing the value of

```
((lambda () (:* (:+ 4 5) (:- 4 5))))
```

. Unstepped tasks are shown in boldface and may be stepped by selecting them with the mouse.

Traveler also provides stepping by transaction, but with a new twist. Steppers usually allow you to step through a procedure call in one step, so you don't have to waste time stepping through procedures

which you think are working fine. Traveler provides this capability by allowing you to step through a transaction in one step, so you can step from a request to its corresponding reply without stepping through the subtransactions needed to compute the reply. The difference is that the transaction is recorded, so that if the reply does not turn out as expected, you can open the transaction and examine its trace to find out what happened.

Displaying the stepped tasks as a trace and stepping by transaction both pose problems in implementation: how do you catch the response for a transaction and display it correctly in the trace? To solve this problem we use the same technique we used in building the trace, the transaction customer. When we step a request, we insert a transaction customer to take care of the reply. The transaction customer keeps track of the correct location in the screen's transaction structure for the reply, so when it receives the reply it is displayed in the correct place ready for stepping.

This stepping facility has proven to be very useful for small and medium small applications, but as we move into larger applications, we have found that there is too much careful but tedious stepping just to get to the problematic parts of the program. A breakpoint mechanism is needed so that we can start stepping in the middle of a program.

B.4 Viewing Lifelines

When there are history sensitive objects in a concurrent computing system, the ordering of events at an object becomes important to observe. We noted above that it is important to be able to control the ordering of concurrent transactions which interact with a common, history sensitive actor; when examining a trace, it sometimes becomes important to know how things happened from a particular actor's viewpoint. For example, there may be many concurrent transactions dealing with the same bank account, and we think we've constrained the ordering of the transactions so that no withdrawals will bounce, but if we find that a transaction occasionally does bounce it is useful to find out in what order the transactions really did arrive at the bank account so we can figure out what went wrong. We record this *arrival order* of recorded tasks at an actor in the actor's *biography*; the display of this biography is called a *lifeline*.

Recording the biography is fairly straightforward; each time we create a task record, we add it to the corresponding target actor's biography. The only subtle point is that we store a copy of the target actor's state in the task record rather than just a reference to the target actor; thus when we look back at the biography we can see a history of states as well as a history of tasks, and better understand why the actor behaved as it did as it evolved into its current state.

Displaying the biography is also straightforward; we display the list of tasks in the order the arrived. Traveler provides a few conveniences: users may change the level of detail with which a task is displayed, or they may hide the events between disparate events in the lifeline to juxtapose them for better comparison. See figure B-6 for an example lifeline.

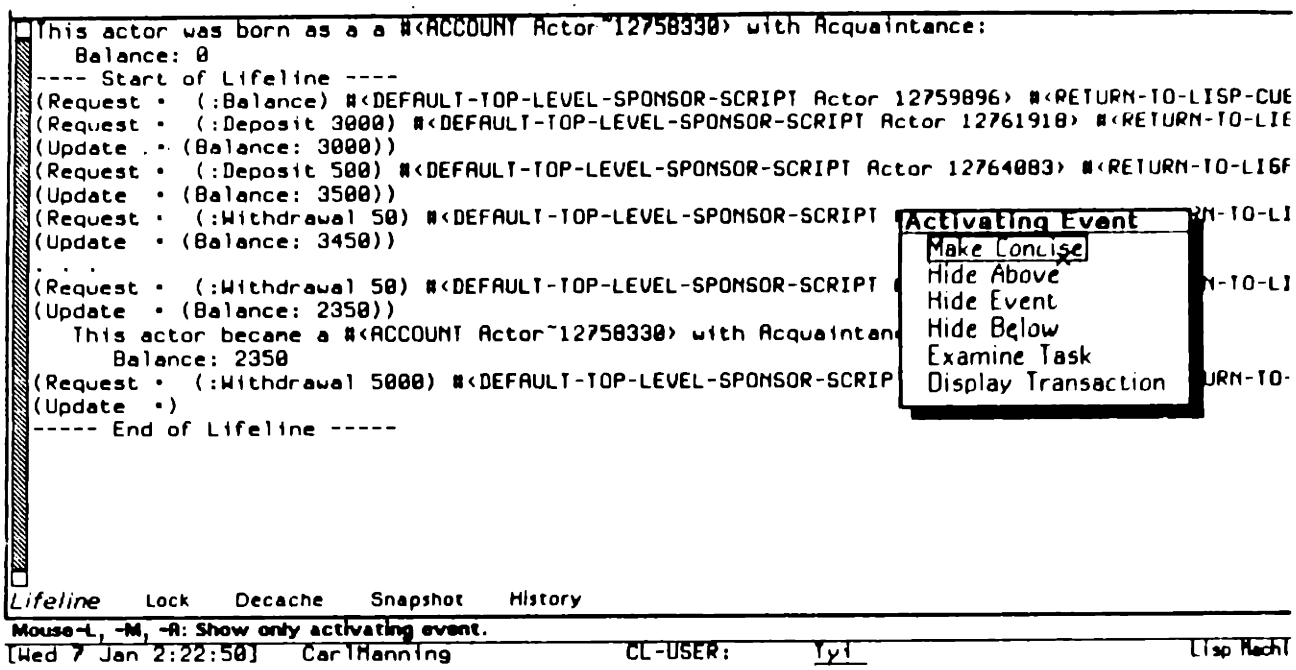


Figure B-6: Lifeline Snapshot

This is the lifeline of a bank account; the ellipses indicate portions of the lifeline are hidden to juxtapose different parts of the lifeline.

In our experience so far, lifelines haven't been used as often as transaction traces or even stepping, but they were very helpful when they were needed. The small programs we've tried so far are probably not complex enough for arrival ordering to get too far out of hand, and I expect lifelines may come into more use as more complex programs are tried. However, lifelines are useful for studying actors which arise as history sensitive managers as programs become more complex. The current format of displaying only the arriving tasks in the lifeline has also been a limitation on their use; it would be a better idea to present a lifeline of transactions where possible so that not only can you find what messages arrived, but you can study what the actor did with each message.

B.5 Summary and Future Work

Traveler is an integrated set of tools for examining actors and actor computations built on a window interface. In addition to the transaction and lifeline displays mentioned here, the current state of an actor may be examined simply by mousing any actor in the display. Traveler's panes can be reconfigured in several sizes to fit the task at hand, and each of the work panes can hold any transaction, lifeline, or examination.

All communication with actors to determine their state and for printing them is done through message passing. However, since debugging tools need to work even when there are problems with locked serialized actors (especially if the locking is the problem), communication with the user's actors is done with special system-requests which bypass the normal locks and handlers. Thus the debugging system bypasses fragile and possibly broken user code while still taking advantage of the distributed message passing nature of the system while it is investigating.

Of course, all this recording does come at a price. While recording, what is normally just two events, a request and a reply, now expands into at least 4 additional events to link the records. Three actors are created: the request record, the reply record, and the transaction customer. Because of this overhead, in our current emulation system programs can take from 5 to 20 times as long to run. Most of this overhead is due to the paging that goes on because of the high memory allocation rate.

Traveler has proven to be a very useful tool for observing and debugging concurrent actor programs on the Apiary. The strategy of recording a concurrent transaction and then examining the record is successful: transaction oriented tracing and stepping have been a huge leap forward over the chronological tracing and stepping facilities which existed before it, allowing us to observe and debug more complex programs. As we develop even larger programs, we are finding a few improvements and additions can be made; in particular, selective tracing and breakpoints are needed.

For the future, we plan to make the improvements I've suggested, as well as work on other tools. In particular, a source-oriented stepper (based on earlier work of Henry Lieberman) is under development. This stepper will display source code and allow concurrent expressions and commands within the code to be stepped by selection; when a value is produced, it replaces the expression which produced it in the code. An interpreter for our core actor language, Acore, is also under development to support this stepper.

Acknowledgments

Many thanks to the members of the Message Passing Semantics Group, who continue to serve as a user community, providing ideas and feedback on Traveler, and especially to Chun Ka Mui, who did much of the early work on Apiary debugging tools.

Appendix C

Acore: An Actor Core Language Reference Manual

Table of Contents

Chapter One

Introduction

Acore is a core actor language for research in actor language development. It is a step above Pract, the primitive actor language, in that the Acore compiler does generate continuation customers for expression evaluation. It also generates the proper customers for joining parallel transactions, either by waiting for all results or by continuing with only the first result. This manual gives an overview of the language, and demonstrates how other language constructs can be quickly implemented in terms of the core constructs through the use of the macro facility.

Overview

The main portion of this manual is concerned with introducing and explaining the various forms which are part of the core language Acore. We start by explaining the top level name scoping and defining constructs, then move on to defining behaviors and actors. This is followed by several chapters presenting the various types of commands and expressions. Finally, there are chapters explaining sponsors (used for computational resource management), complaints (used for signalling exceptions), and interfacing to Lisp (used for I/O and other machine dependent programs).

At the end there are several appendixes which should help you get a little feel for what using this language is like. The appendix on macros shows how many useful constructs can be defined in terms of the core forms. The appendix on compilation gives a few short example programs, and shows how they would be compiled into Pract. It is a good idea to study these examples to understand what is really going on in an Acore program. The final appendix gives a grammar for Acore.

One of these days I hope to insert an introduction to actors here, but I haven't yet, so for now this manual assumes you are familiar with actors and just want to learn about Acore.

Chapter Two

Controlling the Compiling/Loading Name Environment

An Acore program consists of a series of top level forms, each of which is either a definition or a top level command. Definitions are associated with names which may be referenced by other definitions or by top level commands; the subject of this chapter is the management of these names.

There are several levels of names used in Acore. Top level names are those visible at top level and available to the user typing in commands. Modules provide private scoping for the names defined within the module, and export only a few of those names to top level. Inside scripts, which define actor behaviors, names for an actor's acquaintances may be used in any of the script's message handlers. Within each handler, names for message arguments are available, and there may be even smaller scopes binding intermediate values. In this chapter we are concerned with only two types of names associated with definitions, names defined at the top level and names definined within modules.

Top level names provide the working environment for the Acore programmer. Compiling and loading new definitions adds to these names; redefining corrected or updated definitions changes the meaning of these names. A good programming environment for fast prototyping allows the programmer to change the meanings of names easily and quickly; usually this means not taking the excess time to recompile or relink the definitions that use those names. Therefore top level names *indirectly* name the actor defined -- they name a forwarding actor which forwards all its messages to the actor in the definition. Thus redefining a name at the top level only requires changing the forwarding address of the forwarding actor, not of all the actors which use the name. However, this means that the top level name represents a forwarding actor, not the actor produced by the defining expression.

However, this forwarding actor slows down message passing slightly by requiring an extra message pass before the message reaches the actor. Therefore, two methods of defining top level names are provided, one by using DefName to provide the flexibility of indirection through a forwarding actor, and one using DefEquate to provide the extra efficiency of avoiding the extra message pass. Note however that there can be no forward references to actors created using DefEquate, since without the level of indirection it is impossible to create references to an actor before the actor is created.[1]

The following sections describe the Acore forms for declaring new names, declaring module boundaries, and declaring macros to extend the language.

---

[1]Note: This programming environment was designed to provide a programming environment for fast prototyping for research purposes. Systems programming needs may be accomodated by providing a different enviroment, one which accomodates the separate development of modules, and where the top level provides tools for linking the exported and imported names of different modules to produce larger modules.

2.1 Controlling Name Scoping: DefModule

(**DefModule** *[module-name] (recep1 recep2...)* *TOP LEVEL FORM*
top-level-forms...)

DefModule declares the boundaries of modules, which bound the scope of the names declared inside of them. *Module name* is optional; in the future it may be used to name separate modules to be combined to form larger modules. *(recep1 recep2...)* is the list of names declared within the module which will be visibile outside the module; these are the receptionists for the system. Names are declared inside a module by *top level forms*, and may be used by subsequent *top level forms*; any of these top level forms may be nested modules.

Names which are visible in the scope where the module is defined may be used anywhere within the module.

For linking efficiency, DefModule requires that forward references within a module should be declared with DefName first and later redefined. For example:

```
(DefModule quicksort (quicksort)
  (DefName partition nil)
  (DefFunction quicksort (sequence)
    ...
    ... (partition pivot sub-sequence) ...
      ...)
  (DefProcedure partition (pivot sequence)
    ... )
      ...)
```

Within the quicksort module, quicksort makes a forward reference to partition. Without the declaration of partition as a name within this module, partition could be interpreted as a free reference and quicksort might be linked to an actor named partition outside the module. This may be fixed in the future, but it is not a high priority.

2.2 Defining Names: DefName

(**DefName** *name expression*) *TOP LEVEL FORM*

DefName indirectly binds *name* to the value of *expression* in the current environment, be it the top level environment or a module environment. It provides a level of indirection through a forwarding actor so that the definition can be changed without changing all the actors which use *name*; the forwarding actor forwards all its messages to the actor specified by *expression*.

Forward references to names are permitted for names defined with **DefName**. In fact, whenever the loader encounters an undefined name, it assumes it is a forward reference, and creates future for the actor. When the name is finally defined, the future becomes a forwarding actor to the defined value, and any messages which have accumulated are forwarded.

Scripts usually should not be defined using **DefName** since a forwarding actor cannot be used as a

script of an actor. **DefEquate** should be used instead (see below).

2.3 Defining Constants: DefEquate

(DefEquate *name expression***)** *TOP LEVEL FORM*

DefEquate directly binds *name* in the current environment, be it the top level environment or a module environment. It does not provide a level of indirection; therefore, if *name* is redefined, any programs which have used *name* will not see the change unless they are reloaded.

Forward references are not permitted to names defined with DefEquate.

2.4 Defining Macros: DefMacro, DefExpander

Macros allow you to define a syntactic abstraction in Acore. Macro forms are expanded by the compiler into a new form defined by the expansion function. The new form replaces the original form and is compiled.

Acore uses expansion passing style macros [cite], so macro expansion functions are responsible for returning a fully macroexpanded form. To achieve this, expansion functions are passed two extra arguments in addition to the form to be expanded: an expansion function to expand further forms with, and an environment in which macros are looked up. Thus the expansion function has two avenues over which it may change further expansion: either it may supply a different expansion function, or it may supply a different (e.g. augmented) environment.

Macro names have the same lexical scoping properties of other names, so you can define a macro which is only available inside a module, and if you bind a local name the local binding will override any inherited definition of a macro of that name.

At this time the compiler is written in lisp, so the expansion expressions must also be written in lisp.

As with any macro, Acore macros must be defined before they are used, and anything which uses the macro must be recompiled if the macro is redefined.

See the appendix containing possible macros for examples of using DefMacro and some advanced examples using DefExpander.

(DefExpander *name* (*macro-env next-expander form*) *TOP LEVEL FORM*
expansion-expression)

DefExpander defines an expansion function. It is used for complex macros which cannot be defined with the simpler syntax of DefMacro. In particular, it is used for macros where the expander needs to change how sub-forms are expanded, either by augmenting the environment or supplying a

different expander function.

Since expanders are currently written in lisp, there are some lisp auxiliary functions used for manipulating environments and forms. These are **ac:expand, ac:expand-once, ac:map-expander, ac:make-expander, ac:install-expander, ac:install-toplevel-expander, ac:push-macro-environment-frame, ac:install-local-expander, ac:lookup-macro, and ac:initial-expander**. Advanced macro writers may want to look into these and the advanced expander examples in the appendix.

(**DefMacro** *name* (*arg1 arg2...*)
expansion-expression) *TOP LEVEL FORM*

DefMacro defines a simple expansion function which behaves like standard lisp macro expansion functions. The *args...* are bound the parts of the form. The expression returned is sent to incoming the expansion function, so macros may be used in the definitions of other macros.

DefMacro could have been defined as an expander something like the following; note how the incoming expander is called on the results of the expansion:

```
(defexpander DefMacro (macro-env expander form)
  (let ((name (second form)) (args (third form)) (body (cdddr form)))
   `(defexpander ,name (macro-env expander form)
      (let ((expanded-form
              (apply ',(ac:make-expander 'defmacro name args body)
                     form)))
        (funcall expander macro-env expander expanded-form)))))
```

Chapter Three

Defining Actors

To define an actor, you must provide it with a behavior and a set of acquaintances. Behaviors are defined by other actors, called script actors. An actor's state is represented by a tuple of values, its script and its acquaintances. Actors can be created by **create** expressions, but this expression is so primitive that in general functions are provided to create actors. Script actors are created by **script** special form expressions; the major job of the Acore compiler is to convert the behavioral specification given in the **script** form into an executable form.

3.1 Specifying Behavior: Script

(**Script** (*acq1 acq2...*) *SPECIAL FORM EXPRESSION*
[([**:machine-dependent**] [**:actor-types** (*type1 type2...*)])]
communication-handlers...)

A *Script* expression creates and returns an actor representing the general behavior specified by the form. An actor which has this script will implement the behavior, parameterized by its specific acquaintances. That is, it will respond to messages in the ways specified by the communication handlers; these are discussed below.

The **:machine-dependent** option specifies that actors with this script depend upon some aspect of the machine on which they are located (e.g. they are connected to an I/O device) and should not be migrated to other machines. This option is appropriate for any actor which serves as an interface from the actor world to some machine specific hardware. Useful defaults should be defined for script defining macros.

The **:actor-types** (*type1 type2...*) option specifies that actors with this script should be considered to have the types listed. This means that they will answer positively to an **:are-you?** *type* request if *type* is a member of the list. To a **:types** request they will reply with the entire list.

Following the options list are the communication handler specifications. These handlers specify the behavior of an actor with this script to messages with the keyword of the handler. Each type of handler is described below. Note that expression context request handlers expect an expression body, while the other three types of handlers expect commands in their body.

3.1.1 Expression Context Request Handlers

((*:keyword* (*arg1 arg2...*) *COMMUNICATION HANDLER*
*[{***:unserialized** *// ***:serialized** *}]*
*[***:self** *target-symbol]*
*[***:sponsor** *sponsor-symbol]*
*[***:customer** *customer-symbol]*
*[***:reply-keyword** *reply-keyword-symbol])*
expression-body)

The most commonly used type of message handler is a handler for receiving requests and specifying an expression to evaluate and produce a value to return in response to the request. This handler specifies that when it receives a message with selector *:keyword*, the parameters of the message will be bound to the identifiers *arg1 arg2...*, and the expression body will be evaluated. The value produced by the expression body will be sent in a reply message to the customer of the request.

There are several options which can be specified of the handler.

By default, the handler is assumed to be serialized, so a **ready** or **become** command should be performed for every message processed by this handler. The handler may optionally be specified as **:unserialized**, in which case the target is automatically unlocked immediately after receiving the message, and may immediately receive another message. The programmer may also wish to specify that a handler is **:serialized** for documentation purposes.

Also by default, **self** is bound to the target actor which received the request, **sponsor** is bound to the sponsor of the request, **customer** is bound to the customer to which the reply to the request should be sent, and **reply-keyword** is bound to the keyword with which the reply should be labeled. Each of these defaults may be overridden by optionally specifying alternate symbols for the bindings to be overridden.

One request handler may be an otherwise handler which handles any requests which do not match the keywords of the other handlers. The form of an otherwise handler is identical to any other handler except that the *:keyword* is replaced by an unbound indentifier; for example:

((incoming-keyword (&rest args) :unserialized) ...)

The selector keyword of the unmatched message is then bound to the identifier (e.g. **incoming-keyword**) so that it may be forwarded or included in a complaint message.

The expression context request handler expects an expression body. An *expression body* is a list of one or more commands or expressions, the last of which must be an expression. The value of the expression body is the value of the last expression.

3.1.2 Command Context Request Handlers

(Is-Request (*:keyword* (*arg1 arg2...*) *COMMUNICATION HANDLER*
*[{***:unserialized** *// ***:serialized** *}]*
*[***:self** *target-symbol]*
*[***:sponsor** *sponsor-symbol]*

[**:customer** *customer-symbol]*
[**:reply-keyword** *reply-keyword-symbol])*
command-body)

This handler is identical to the expression context request handler except that its body provides a command context rather than an expression context. This handler is used in situations where no reply value is to be specified, for example if the request must be put into a waiting queue for later processing which will then produce a reply.

A *command body* is a list of commands or expressions. Since it is a command body, it is treated as a command and has no value. The values of any expressions of the command body are ignored. Any response to the request must be produced in some other way. Two possible ways of returning a response to the customer are explicitly producing a response with the **reply-to** or **complain-to** commands, or by forwarding a request to another actor using a **request** command with the same customer.

3.1.3 Command Context Reply Handlers

(**Is-Reply** (*:keyword* (*arg1 arg2...*) *COMMUNICATION HANDLER*
*[{***:unserialized** *// ***:serialized** *}]*
[**:self** *target-symbol])*
command-body)

This handler is identical to the command context request handler except that it handles reply messages. This handler is very rarely used; it is only used when a special behavior is needed for a customer which the compiler cannot produce. Normally the compiler generates all the customers needed while compiling **ask** expressions and **lets**, so if you are using this you are probably doing something wrong.

The fact that the incoming message is just a reply rather than a request makes several differences. There is no default customer or reply keyword in the incoming message, so there is no form of the reply handler with an expression body. There is no default sponsor in the incoming message, so any processing done in the command body of the reply handler must be wrapped in a **with-sponsor** form. Since there is no incoming request from which to bind a sponsor, customer, or reply-keyword, there are no default bindings and no options for specifying a binding for any of these.

3.1.4 Command Context Complaint Handlers

(**Is-Complaint** (*:keyword* (*reply-keyword arg1 arg2...*) *COMMUNICATION HANDLER*
*[{***:unserialized** *// ***:serialized** *}]*
[**:self** *target-symbol])*
command-body)

This handler is identical to the command context reply handler except for two things: it handles complaint messages instead of replies, and because of this it must have at least one parameter in the argument list to bind the reply keyword. Like reply handlers, complaint handlers are used only when a special customer must be specified, so if you are using this you are probably doing something wrong.

Normally complaint handling can be specified using **let-except**.

Complaint messages are responses aborted transactions. The keyword of a complaint message gives some indication as to why the transaction occurred. The reply keyword is also included so that the customer of the transaction can identify the transaction (joining customers may be the customer for several concurrent transactions). The reply keyword is only bound, rather than used in choosing the handler, because the arguments of the complaint will depend upon the type of the complaint, which is distinguished by the *:keyword*.

It is especially important to specify an **is-complaint** otherwise handler for any actor which may receive responses so that any unforseen complaints generated by errors will be trapped and taken care of.

3.1.5 Handler Parameter Lists

The parameter list to any handler may include the lisp-like keywords **&key**, **&allow-other-keys**, **&rest**, and **&optional**. **&key** means that the arguments following it are keyword arguments, allowing named rather than positional notation for matching formal to actual arguments. Here is an example in both positional an keyword notation:

```
...((:do (price size color))...)
        Invoked by: (:do order p s c)
...((:do (&key price size color))...)
        Invoked by: (:do order :color c :price p :size s)
```

&allow-other-keys means that extraneous keywords should not cause an error, but should be ignored.

&rest means any further actual arguments will be combined into a special list and bound the following formal argument. This is useful for forwarding arbitrary messages, e.g.

```
...(is-request (incoming-keyword (&rest arguments) :unserialized)
      (request* forwarding-address (incoming-keyword arguments)
            sponsor customer reply-keyword))...
```

&optional means the following arguments may be omitted from the actual arguments. If so, the formals will be bound to nil.

Here is an example Script expression, in this case for a forwarding actor which forwards all its messages to the number 0.

```
(defname forwarding-actor
  (create
   (script (forwarding-address)
       (:actor-types (forwarding-actor))
     (is-request (incoming-keyword (&rest arguments) :unserialized)
       (request* forwarding-address (incoming-keyword arguments)
           sponsor customer reply-keyword))
     (is-reply (incoming-keyword (&rest arguments) :unserialized)
       (reply-to* forwarding-address
           (incoming-keyword arguments)))
     (is-complaint (incoming-keyword (reply-keyword &rest arguments)
```

```
                    :unserialized)
      (complain-to* forwarding-address
          (incoming-keyword reply-keyword arguments))))
0))
```

3.2 Creating New Actors: Create

(Create *script-expression acquaintance-expressions...*) *EXPRESSION*

Create is a low level primitive for creating new actors. A create expression returns a new actor with the script specified by *script-expression* and with the acquaintances specified by the *acquaintance expressions*. It performs no checking that the types or even number of acquaintances are appropriate for the script.

Script-expression must evaluate to a script actor; a forwarding actor to a script actor will not do in this case.

Because of the low level nature of the **create** expression, it is strongly recommended that any abstractions which require creating actors provide functions for doing the creation. These creation function should perform any number and type checking necessary. See the macros for DefBehavior and DefStructure in the Macros appendix.

Here is an example Create expression, using the forwarding script defined above. Assume that **new-address** is defined. This expression creates a new forwarding actor with **forwarding-script** as its script and new-address as its single acquaintance (the forwarding-address).

```
(create forwarding-script new-address)
```

Chapter Four

Message Passing Expressions and Commands

4.1 Ask Expressions

(*:keyword target arguments...*) *EXPRESSION*

An ask expression sends a request to an actor, and its value is the response returned as a result of the request. The request is sent to the value of the *target* expression. The request is specified by the value of the *:keyword* expression, which should evaluate to a keyword, and parameterized by the values of the *argument* expressions. Parameters are passed strictly by value (but see **future** and **delay** expressions for simulating concurrent and delayed evaluation).

4.2 Call Expressions

(*function arguments...*) *EXPRESSION*

A call expression is simply shorthand for the ask expression:

(**:do** *function arguments...*)

Functions follow the convention of expecting :do keywords; see DefFunction in the Macro appendix.

Keywords such as **:deposit** may also serve as functions; what they do is to send the first argument a message with themselves as the selector and pass on any further arguments. So for example, if **deposit** is bound to **:deposit**, then

```
(deposit account 10)
```

is equivalent to

```
(:do :deposit account 10)
```

and when **:deposit** receives the **:do** message, it in turn sends the message

```
(:deposit account 10)
```

.

4.3 Returning Multiple Values

values *values to be returned...*

Normally an expression returns a single value. The **values** expression may be used to return multiple values for an expression. The multiple values *must* be bound with a **let** statement which handles them; an expression returning multiple values cannot be used as a subexpression. Binding multiple values is covered under **let**.

4.4 Request Commands

(**Request** *target* (*:keyword arguments...*) *COMMAND*
sponsor customer reply-keyword)

A **request** command sends a request to the actor specified by the target expression. The request contains the keyword and arguments specified by the *:keyword* and *argument* expressions. The transaction will be sponsored by the actor specified by the *sponsor* expression. The response will be sent to the actor specified by the *customer* expression, and if it is a reply it will carry the keyword specified by the *reply-keyword* expression.

Request commands need to be used only when programmers want to specify their own customer for some reason; usually it is much easier to let the compiler generate the customers by using ask expressions.

4.5 Reply-to Commands

(**Reply-to** *target* (*:keyword arguments...*)) *COMMAND*

A **reply-to** command sends a reply to the actor specified by the *target* expression. The reply contains the keyword and arguments specified by the *:keyword* and *argument* expressions.

Reply-to commands need to be used only when the programmer needs to explicitly delay the reply for some reason, and reply at some later date. Usually it is much easier to use an expression context request handler which generates the reply command automatically.

4.6 Complain-to Commands

(**Complain-to** *target* (*:keyword reply-keyword arguments...*)) *COMMAND*

A Complain-to command sends a complaint to the actor specified by the *target* expression. The complaint contains the keyword, the reply-keyword, and arguments specified by the *:keyword reply-keyword* and *argument* expressions.

The *:keyword* identifies the type of complaint. The *reply-keyword* must be included to identify the transaction which complained.

Complain-to need to be used only in contexts similar to reply-to commands. Usually it is much easier to use an expression context request handler and a **complaint** expression, which will generate the complaint automatically.

4.7 Request*, Reply-to*, Complain-to*

(**Request*** *target* (*:keyword argument-list*) *COMMAND*
sponsor customer reply-keyword)
(**Reply-to*** *target* (*:keyword argument-list*)) *COMMAND*
(**Complain-to*** *target* (*:keyword reply-keyword argument-list*)) *COMMAND*

These forms of the message passing commands are useful for forwarding messages. *argument-list* must be a special list bound by an **&rest** argument. See the forwarding actor script at the end of the **script** section for examples of using these commands.

4.8 Ready

(**Ready** *{(acquaintance-name new-value)}**) *COMMAND*

Upon receiving a communication with a serialized handler, an actor remains locked and may not process another communication until a **ready** or **become** command is processed, specifying its new state and releasing the lock. A ready command without any arguments simply releases the actor to process its next message. A ready command with (*acquaintance-name new-value*) pairs updates the specified acquaintances with the new values; these are the values the acquaintances will have during the processing of the subsequent message. For example, a bank account with an acquaintance called **balance** might have a handler for deposit requests which looked like this:

```
(script (balance) ...
...((:deposit (deposit-amount))
    (let ((new-balance (balance :+ deposit-amount)))
      (ready (balance new-balance))
      ':ok))...)
```

Note that *acquaintance-name* must be the literal name, and not an arbitrary expression which evaluates to the name.

4.9 Become

(**Become** *new-script new-acquaintances...*) *COMMAND*

A become command is used when an actor completely changes its state, for example when an actor representing a future or a thunk (delayed expression) becomes a forwarding actor to the resulting value. In this case, a new script must be specified, as must all the new acquaintances needed for the new script. Like create, this command is dangerous because there is no checking that the types of acquaintances or their number agrees with the requirements of the script.

4.10 Replace

(**Replace** *target new-script new-acquaintances...*) *COMMAND*

Replace is a primitive used to completely change the state of a locked actor. It differs from **become** in that the target actor is specified. Like **create**, this is a primitive and it is *strongly*

recommended that it be encapsulated in a function which can check that the number and type of the acquaintances agree with the script.

Chapter Five

Controlling Subexpression Evaluation

This chapter introduces the forms **future**, **delay**, and **race**. Futures allow expression values to be passed around concurrently with the evaluation of the expression. Delays allow the evaluation of an expression to be delayed until the actual value is needed; it may be passed around but isn't evaluated until it is sent a message. Race performs the evaluation of its subexpressions concurrently, but it only waits for the first result to return; this is the result of the race expression.

5.1 Future

(Future *expression*) *EXPRESSION*

(Future)

A future expression immediately returns a future actor which represents the value of the expression. Simultaneously, the evaluation of the expression is begun, with the future as the customer to which the value will ultimately return. While the future is waiting for the result of evaluating the expression, it buffers any requests it may receive. When it receives the result of the expression, if it is a valid reply then all the buffered requests are forwarded to the value and future becomes a forwarding actor to the value. If instead a complaint had been generated, then the future becomes a complaint generator, processing all the buffered requests by complaining to the customers and returning complaints to any new requests it receives.

A future expression without *expression* just returns a future. The programmer is expected to provide a replacement behavior (e.g. by calling a function which issues a replace command) for it sometime in the future; otherwise it will just buffer messages forever.

5.2 Delay

(Delay *expression*) *EXPRESSION*

A delay expression immediately returns a delay actor which represents the expression and the closure over the environment. Unlike the future actor, the delay actor remains dormant until it is sent a message. Upon receiving a message, the delay actor lets the expression evaluate. Upon receiving a result, if it is a valid reply then the message is forwarded to the value and the delay actor becomes a forwarding actor to the value. If instead the result was a complaint, then the delay actor returns a complaint to the customer of the request and becomes a complaint generator just like the future.

5.3 Race

(Race *expressions...*) *EXPRESSION*

A race expression concurrently begins the evaluation of all its subexpressions. It returns an actor which behaves as an input stream; the values of the expressions are stored in the stream in the order that they arrive. These values can be extracted from the stream by sending it successive `:extract` messages.

The stream only stores successful values; complaints are not stored in the stream. Thus the stream stores values in the order of successful completion. Complained values can be stored in the stream by wrapping each of the argument expressions in a **let-except** form; the value returned by this form will then be the value of the expression and, if it is not a complaint, will be stored in the stream.

Once the stream has returned all successful values, it will return `:end-of-stream` to all successive `:extract` messages.

Chapter Six

Binding Intermediate Values

This chapter introduces the **let** command and expression. Let forms allow a single or multiple values in replies to be bound, just as request handlers may bind multiple argument values.

6.1 Let

(**Let** (*{ ((val1 val2...) expression) }**) *EXPRESSION*
expression-body)
(**Let** (*{ (val expression) }**) *EXPRESSION*
expression-body)
(**Let** (*{ ((val1 val2...) expression) }**) *COMMAND*
commands...)
(**Let** (*{ (val expression) }**) *COMMAND*
commands...)

The multiple value form of a **let** binds the identifiers *val1 val2...* to multiple values returned from *expression*. The identifier list can take any of the forms of a handler parameter list (minus the leading selector keyword), including the use of **&key**, **&allow-other-keys**, **&optional**, and **&rest**. (See the section on handler parameter lists for an explanation of these.) If a parameter mismatch occurs, a complaint will be generated.

If only a single value is expected, then the single value form of **let** may be used for brevity. A single symbol in a let arm binding is equivalent to a list of the symbol. For example, the following two let forms are equivalent:

```
(let (((total) (sum :do sequence))
      ((count) (sequence :length)))
  (total :/ count))
(let ((total (sum :do sequence))
      (count (sequence :length)))
  (total :/ count))
```

If there are several arms with expressions to be bound, the expressions are evaluated concurrently; when all the expressions have been evaluated, then the forms of the body are performed concurrently in the environment extended by the new bound identifiers. If any of the expressions results in a complaint, the result of the let form is the complaint and the body is not performed. Sequential binding can be accomplished by nesting let commands or expressions.

Examples of use:

```
(let (((first rest) (cons-cell :decompose)))
  (if (null? rest)
      (then first)
      (else rest)))

(let (((&key ((:name name1)) ((:number number1))
             ((:quantity quantity1)) ((:price price1)))
       (order1 :keyword-decompose))
      ((&key ((:name name2)) ((:number number2))
             ((:quantity quantity2)) ((:price price2)))
       (order2 :keyword-decompose)))
  ((price1 :* quantity1) :+ (price2 :* quantity2)))
```

Chapter Seven

Flow of Control

The **if** command is the sole core primitive provided for making decisions and controlling the flow of the computation. Other constructs can be made with macros, building on the primitive capability provided by **if**.

Without message passing, an actor can only test whether two mail addresses are identical, and form **and**, **or**, and **not** combinations of such tests. Therefore, any message passing expression used as the conditional part of an **if** form is interpreted as a comparison between two mail addresses: the address returned by the expression and the address of the unique actor representing falsity, **nil**. Thus, *any programs which use futures, delays, and other forwarding actors must be careful about what values are compared and used as conditions.* In particular, a forwarding actor (future, delay) which forwards to **nil** will not be considered **nil**, and will always be interpreted as a true value rather than false. This is because the forwarding actor is an actor in its own right with its own mail address, which is distinct from the mail address of every other actor, including the **nil** actor.

Therefore it is imperative that when using futures and delays, the address of the expression value rather than the forwarding actor be used in comparisons and conditions. The value can be obtained by sending a **:self** message to the actor to be compared; the forwarding actor will forward this message, and non-forwarding actors reply with their mail address in response to this message. See the examples below.

7.1 If

(If *test-expression* — *SPECIAL FORM EXPRESSION*
  (then *expression-body*)
  (else *expression-body*))

(If *test-expression* — *SPECIAL FORM COMMAND*
  (then *commands...*)
  (else *commands...*))

An **if** form evaluates and tests the value of *test-expression*; if the value is not the actor representing falsity, **nil**, then the **then** branch is performed; otherwise the **else** branch is performed. The value of an **if** expression is the value of the *expression-body* performed. If evaluating *test-expression* results in a complaint, neither branch is performed and the result of the **if** form is the complaint.

Examples of use:

```
(If 'true
   (then ':yes)
   (else ':no)) --> :yes
(If nil
   (then ':yes)
   (else ':no)) --> :no

(if (future nil)
   (then ':yes)
   (else ':no)) --> :yes
(if ((future nil) :self)
   (then ':yes)
   (else ':no)) --> :no

(if (delay nil)
   (then ':yes)
   (else ':no)) --> :yes
(if ((delay nil) :self)
   (then ':yes)
   (else ':no)) --> :no
```

Chapter Eight

Primitive Expressions

Primitive expressions in Acore are those expressions which do not require any further message passing. They include constants, strings, bound identifiers, the identity (==) predicate, and the boolean operators **not**, **or**, and **and**.

8.1 The Identity Predicate

(== *expression expression*) *EXPRESSION*

As mentioned in the previous chapter, the primitive decision making ability of actors is based on comparing whether two mail addresses are identical. The identity predicate performs this comparison, and decisions are made based on whether the mail addresses produced by the two examples are identical or not. You may think of the conditional expression of an `if` statement as testing whether the result is identical to the **nil** actor. For example,

```
(if (actor :are-you ':integer)
  (then ...)
  (else ...))
```

is interpreted as

```
(if (== nil (actor :are-you ':integer))
  (then ...)
  (else ...))
```

However, to increase the flexibility of using this predicate as an arbitrary expression, it can also be used to return a value. For example,

```
(ready (:acq (== actor unique-value)))
```

is equivalent to the following

```
(ready (:acq (if (== actor unique-value)
                 (then t)
                 (else nil))))
```

8.2 Boolean Expressions

The following boolean expressions have much the same meaning that they do in Pract. They interpret **nil** as false and any other value as true; **not** returns **t** as true. (This may change if we decide there should be specific boolean actors rather than using symbols.)

8.2.1 Not

(**not** *expression*) i{EXPRESSION

The value of a **not** expression is a truth value (t) if the value of *expression* is **nil**; otherwise the value is **nil**.

8.2.2 And

(**and** *expressions...*) *EXPRESSION*

The value of a **and** expression is **nil** if the value of any of the *expressions* is identical to **nil**; otherwise the value is the value of the last expression. Note that all expressions are evaluated.

8.2.3 Or

(**or** *expressions...*) *EXPRESSION*

The value of an **or** expression is **nil** if the value of all of the *expressions* is identical to **nil**; otherwise the value is the value of the first non-nil expression. Note that all expressions are evaluated.

8.3 Bound identifiers

identifier *EXPRESSION*

The value of a bound identifier is the actor which is bound to it. Identifiers may be bound by virtue of being bound in the external environment, by being bound in the local module environment, by being an acquaintance of the actor, by being a formal argument for a message handler or complaint handler, or by being bound by a **let** statement.

8.4 Literals

There are four forms of literals: keywords, numbers, strings, and quoted list structure.

8.4.1 Keywords

:keyword *EXPRESSION*

A keyword is a symbol preceded by a colon, and evaluates to itself. Keywords are unique; two keywords with the same spelling must be identical, since keywords are the basis of message handler selection.

8.4.2 Numerals

numeral *EXPRESSION*

Numerals are strings of digits, possibly with a leading sign, possibly with a single decimal point. A numeral evalutes to the number it represents; numbers are **not** necessarily unique, depending upon the implementation, so identical numerals may or may not evalutate to identical numbers.

8.4.3 Strings

"string" *EXPRESSION*

A string is a sequence of characters surrounded by double-quotes. To include a double quote or a backslash in a string, it must be preceded by a backslash.[2] It evalutates to an actor which represents the sequence of characters.

8.4.4 Quoted List Structure

'list-structure *EXPRESSION*

Quoted list structure evaluates to the list structured under the quote, unevaluated. At this time it may or may not evaluate to an identical actor each time.

[2]Backslash is the Common Lisp single escape character.

Chapter Nine

Sponsors

This chapter introduces the concept of sponsors and the constructs in Acore for using sponsors. There is also a message protocol which is, so far, underdeveloped.

In a processing environment where there may be many threads of control proceeding concurrently, frequently forking more threads or dying, there is a need to control the rate at which each of these threads proceeds, possibly aborting it entirely if it becomes unnecessary or wasteful. It is the role of actors acting as *sponsors* to control the processing rates of the computations under its control.

Each transaction (a request and response) in an actor computation must be sponsored. When the transaction is run, the system requests *ticks* from the sponsor, using the **:more-sponsor-ticks** message. The sponsor may either reply with some number of ticks, or it may respond with the complaint **:sponsorship-denied** meaning that the sponsor is unwilling or unable to grant any more ticks and thread must be aborted. If the transaction is aborted, it will return a **:sponsorship-denied** complaint, so any serializers can be unlocked, undoing any state changes in progress.

If a sponsor runs out of ticks, it may ask its parent sponsor for more ticks. If all the computations controlled by a sponsor become unnecessary, the sponsor may be sent a **:stifle** message to abort all its computations; it should respond with any remaining ticks.

These protocols will be extended. It needs protocols for temporarily suspending a computation if the sponsor wants to hold it up for a while -- just not replying for a while is not good enough because the computation may be holding some locks or resources which are needed by other computations. It also needs a protocol for guaranteeing a critical computation can complete without running out of ticks. A protocol for letting a sponsor know when a computation it is sponsoring has completed would also be useful.

9.1 Sponsor Messages

9.1.1 :More-Sponsor-Ticks

:more-sponsor-ticks *max-allowed* *REQUEST MESSAGE*

Sponsors receive a **:more-sponsor-ticks** message from the system when a transaction needs to be sponsored. Rather than ask for sponsorship for every transaction, the system requests ticks for many transactions. *Max-allowed* is the maximum number of ticks the system will accept at once; any more may be thrown away. Sponsors should either reply to this message with an integer number of ticks

granted, or complain **:sponsorship-denied** to abort the transaction.

9.1.2 :Stifle

:stifle *REQUEST MESSAGE*

Sponsors may be sent a **:stifle** message if all the computations they are sponsoring should be aborted. For example, after receiving a **:stifle** message, a sponsor may respond to all requests for more ticks with the complaint **:sponsorship-denied**. (Note however that this is not required; in fact sponsors sponsoring critical transactions which must be run to completion should not do this, but should continue sponsoring the transaction to completion with the sponsor's current tick supply.) Sponsors should respond to the **:stifle** message by returning unused sponsor ticks.

9.1.3 :Sponsorship-Denied

:sponsorship-denied *COMPLAINT MESSAGE*

Sponsors return the complaint **:sponsorship-denied** if they are unwilling or unable to grant more ticks. A transaction which was aborted because sponsorship was denied returns this complaint as well.

9.2 Sponsor Forms

9.2.1 With-Sponsor

(With-Sponsor *sponsor-expression body-expression*) *EXPRESSION*

(With-Sponsor *sponsor-expression commands...*) *COMMAND*

With-sponsor first evaluates the sponsor expression. This actor then sponsors the performance of the commands and expressions of the body.

Chapter Ten

Complaints

Complaints are the exception signalling mechanism of Acore. They are messages like replies, but they are handled by different handlers. Complaints are generated by the **complaint** form and the **complain-to** command. If an expression generates a complaint, either because the response to an ask was a complaint message or because a **complaint** expression was encountered, the handling of the complaint is specified by the closest lexically enclosing **let-except** which handles the complaint generated. It is an error for a complaint to be generated in a context with no exception handler to handle it, so most contexts should provide at least a default otherwise exception handler. For example, the default request handlers with expression bodies provide default exception handling; the expression context it provides is one place where a default (forward the complaint to the customer) makes sense.

10.1 Complaint Forms

10.1.1 Complaint

(Complaint *keyword-describing-the-complaint* *EXPRESSION*
other-arguments-giving-context...)

Complaint generates complaints in expression contexts; the evaluation of the expression is aborted and the enclosing complaint handling invoked. *Keyword describing the complaint* is a keyword by which the complaint handler will be chosen. *Other arguments giving context* is usually a list of keyword arguments giving the actors which were involved, for example the target actor which generated the complaint, the message which it received, the customer which expected the reply (and from which the sender of the message can usually be deduced), and any other information which may be pertinent depending upon the exception. See messages below for examples. These keyword arguments should probably be standardized, but this hasn't been done yet.

A complaint expression which is not enclosed by a Let-Except form providing a handler for it is illegal. See Let-Except for an example of the valid use of complaint and let-except.

10.1.2 Let-Except

(Let-Except (*{ ((val1 val2...) expression) }**) *EXPRESSION*
(Except-When
{ ((:keyword (valA valB...)) expression-body) })
expression-body)
(Let-Except (*{ (val expression) }**) *EXPRESSION*
(Except-When
{ ((:keyword (valA valB...)) expression-body) })

expression-body)

(**Let-Except** ({ ((*val1 val2...*) *expression*) }*) *COMMAND*
  (**Except-When**
    { ((*:keyword* (*valA valB...*)) *commands...*) })
  commands...)

(**Let-Except** ({ (*val expression*) }*) *COMMAND*
  (**Except-When**
    { ((*:keyword* (*valA valB...*)) *commands...*) })
  commands...)

Let-Except, when used in conjunction with the **complaint** expression within a script and complain-to commands in other scripts, provides a means of non-local exit for exception handling. A **let-except** form is evaluated like **let** -- the expressions in the arms are concurrently evaluated, the results are bound to identifiers, and the body is performed in the environment resulting from extending the surrounding environment with the new bindings -- **except when** one of the expressions in the arms generates a complaint. In this case the enclosing exception handler with a matching keyword is invoked, its parameter list is bound to the parameters of the complaint, and the body of the exception handler is performed in the environment resulting from extending the environment surrounding the let-except form with the bindings of the parameter list. In the case of the let-except expression, the value of the handler's body becomes the value of the let-except expression.

Note that the scope of the exception handlers is limited to the arms of the let-except statement, and not the body. This design allows the same exception handlers to cover one or several concurrent expressions.

Like message handlers, one of the exception handlers in a clause may be an otherwise handler. This is indicated by replacing the keyword with a normal symbol, and the keyword will be bound to that symbol in the body of the handler.

Note that the environment within the exception handlers does not include the bindings of the arms.

An example of how **complaint** and **let-except** may be used:

```
(DefFunction make-account (account-no name password initial-balance)
  (Let-Except
       ((record
          (Let-Except
               ((account (create account-script account-no
                                 ; If bad password, abort
                                 (if (:> 5 (:length password))
                                     (then (complaint :bad-password
                                                      password))
                                     (else password))
                                 ; If no funds, trap
                                 (if (:<= initial-balance 0)
                                     (then (complaint
                                             :insufficient-balance))
                                     (else initial-balance)))))
                ; Catch Trap if no funds -- make a different record
```

```
            (Except-When
             ((:insufficent-balance)
              (make-waiting-for-deposit-record
                 name password account-no)))
          ; Otherwise make a normal record
          (make-record name account))))
      ; Catch any other traps, and complain to customer.
      (Except-When
       ((some-error (&rest args))
        (complain-to* customer (some-error reply-keyword args))))
    ; If all is normal, just return the record.
    record))
```

This might be an Acore implementation of a function which opens accounts. It normally creates a record with the name and the account, and the account itself, which holds the balance and the password. In creating the account, it may notice anomalous conditions uses the complaint mechanism to issue exit out to a trap which performs the job in light of the condition. An `:insufficent-balance` complaint is caught by the inner exception handler, which has a matching pattern; this handler resolves the situation by creating a different kind of record instead, which is returned normally. Any other complaint which may be issued, such as if the password didn't recognize the `:length` message and complained, or the **:bad-password** complaint was issued, will invoke the otherwise handler (`some-error...`) and complain to the customer.

10.2 Some Complaint Messages

Below are some of the complaint messages generated by the system.

10.2.1 Unrecognized-Request

:unrecognized-request *COMPLAINT MESSAGE*
:target *actor which received the request*
:sponsor *sponsor sponsoring the request*
:customer *customer of the request*
:reply-keyword *reply-keyword of the request*
:selector *selector keyword of the request*
:arguments *list of other arguments to the request*

This complaint is generated when an actor receives a message for which it has no handler, i.e. the selector keyword does not match the keyword of any of its handlers.

10.2.2 Machine-Error-Trap

:machine-error-trap *COMPLAINT MESSAGE*
:target *actor which received the request*
:sponsor *sponsor sponsoring the request*
:customer *customer of the request*
:reply-keyword *reply-keyword of the request*
:selector *selector keyword of the request*
:arguments *list of other arguments to the request*

:report *string describing problem*

The system generates this complaint when a firmware error is trapped. Many errors, such as arithmetic errors, wrong number of arguments, etc., will initially generate this type of error, but hopefully many of these will someday generate a unique complaint which can be recognized.

Chapter Eleven

Interfacing with Lisp Programs and I/O Devices

Since Acore will initially be implemented on systems supporting Lisp, a mechanism to interface with the Lisp system is provided so that Acore programs can interact with existing Lisp software, especially I/O systems such as the file system and the window system.

11.1 Interfacing to Lisp from Acore

References to lisp functions and other object cannct be migrated from machine to machine as Acore objects; Lisp assumes that all objects it operates on are local. Therefore any actors which reference Lisp objects should be declared **:machine-dependent** (see **script**) so they will not be migrated. For example it doesn't make sense to display a string for the user on a machine other than the one connected to the user's console, so the actor which represents any window must be declared machine dependent. Since the **:machine-dependent** declaration only prevents migration of actors, you should also be careful to make sure the actor is created on the correct machine; this may mean making its creator machine dependent as well.

11.1.1 The #L Form

#L(*lisp-function-symbol arguments...*)

SPECIAL FORM EXPRESSION }

The #L form first evaluates the arguments of the lisp function call as Acore expressions. If no complaints occur, a separate process is created by the Lisp system on the current machine, and the lisp function *lisp function symbol* is called with the arguments in that process. If the function returns normally, the value returned is the value of the expression. If the function returns an error, a **:machine-error-trap** complaint is generated describing the error.

Note that because **lisp-function-symbol** is looked up in the global Lisp environment of the machine on which it is invoked, actors which invoke it should be machine dependent. Details of how the argument objects/actors will appear to the Lisp function are implementation dependent, but probably the representation of integers, symbols, and strings will be common, and at least a conversion to other list objects, such as lists, will be available.

11.2 Interfacing to Acore from Lisp

Interfacing to Acore from Lisp consists of making requests to Acore actors from Lisp, and receiving the reply.

11.2.1 Ask-From-LIsp

(ask-from-lisp-code *target sponsor :keyword arguments...*) *LISP FORM*

The Lisp function **ask-from-lisp-code** makes a request to the Acore actor specified by the lisp expression *target*. The request will be sponsored by the Acore actor specified by *sponsor*. The message will contain *:keyword* and any *arguments* supplied. This function creates a machine dependent actor as the customer for the request, then it creates a task sending the message to the target and puts it on an Apiary queue to be executed. Finally it suspends the lisp process. When a response returns, it will be sent to the customer, which communicates the results to the Lisp process and awakens it. Finally, the function returns the result of the request.

If a single value is returned to the customer, then this value is returned by **ask-from-lisp-code**. If a complaint is returned by the customer, or a multiple value reply is returned, then **ask-from-lisp-code** generates a Lisp error.

Appendix A

Macros for Advanced Constructs

11.3 Return

Return is a very simple macro for the common instance of replying to the default customer with the default reply-keyword in command context.

```
(DefMacro Return (expr)
  `(reply-to customer (reply-keyword ,expr)))
```

11.4 Block

Block is a very simple macro for introducing an expression body in expression context.

```
(DefMacro Block (&rest expression-body)
  `(let () ,@expression-body))
```

11.5 Lambda

Lambda creates and returns a function actor by creating a script which defines the behavior of the body and creating and returning the actor which has this script.

```
(DefMacro Lambda (arguments &body body)
   `(create (script () (:actor-types (lambda))
                    ((:do (,@arguments) :unserialized)
                       ,@body))))
```

Example of use:

```
(map (lambda (starting-balance)
        (make-bank-account starting-balance))
      starting-balances)
```

11.6 DefFunction

DefFunction is a form for defining function actors, actors with no local state who return a result when called with a set of arguments. They may be called with the form [*name args...*] which is equivalent to the form (*name* :do *args...*). Since functions don't change state, it is declared unserialized.

```
(DefMacro DefFunction (name args &rest body)
  `(Defname ,name (lambda ,args ,@body)))
```

Example of use:

```
(DefFunction factorial (n)
  (if (:= n 0)
      (then 1)
      (else (:* n (factorial (:- n 1))))))
```

11.7 CLambda

CLambda is much like lambda except that it produces an actor with a command body rather than an expression body.

```
(DefMacro CLambda (arglist &rest body)
  `(create (script ()
             (is-request (:do (,@arglist) :unserialized)
               ,@body))))
```

11.8 DefProcedure

DefProcedure is much like DefFunction except that it provides a command body rather than an expression body.

```
(DefMacro DefProcedure (name arglist &rest body)
  `(defname ,name (clambda ,arglist ,@body)))
```

11.9 SScript

A macro for safe scripts creates a guardian for the script which handles :create and :replace messages.

```
(DefMacro SScript (acqs &body handlers)
  `(let ((the-script (script ,acqs ,@handlers)))
     (create (script ()
               ((:create ,acqs :unserialized)
                (create the-script ,@acqs))
               ((:replace (target ,@acqs) :unserialized)
                (replace target the-script ,@acqs)
                target)))))
```

Example of use:

```
(DefName bank-account-script
 (SScript (balance)
  ((:balance () :unserialized)
   balance)
  ((:deposit (deposit-amount))
   (let ((new-balance (:+ balance deposit-amount)))
     (ready (balance new-balance))
     ':ok))
  ((:withdrawal (withdrawal-amount))
   (let ((new-balance (:- balance withdrawal-amount)))
     (if (:< balance 0)
       (then (ready)
             (complaint :overdraft
                        :by-amount new-balance))
       (else (ready (balance new-balance))
             withdrawal-amount)))))
(defname my-account (:create bank-account-script 1000000))
```

11.10 DefStructure

DefStructure is a macro for making simple actors which just store values in their slots, allowing you to send messages to retrieve or set their values. A similar macro could be made for immutable structures by omitting the set handlers. Structure actors also respond to a :decompose message which replies with all the components, so they may be taken apart with a single message.

```
(DefMacro DefStructure (name slots)
 `(DefName  ,name
    ((SScript ,slots
      (:actor-types (,name))
      ,@(map (lambda (slot-name)
               `((,(make-keyword slot-name) ())
                 ,slot-name))
             slots)
      ,@(map (lambda (slot-name)
               `((,(make-keyword
                          (symbol-append 'set- slot-name))
                  (new-value))
                 (ready (,(make-keyword slot-name) new-value))
                 self))
             slots)
      (is-request (:decompose)
        (reply-to customer
          (reply-keyword
            ,@(flatten
                (map (lambda (slot-name)
                       (list (make-keyword slot-name)
                             slot-name)) slots)))))))))
```

Example of use:

```
(DefStructure cons-cell (car cdr))
(DefFunction cons (car cdr)
  (:create cons-cell car cdr))
```

11.11 Sequential Expressions

Sequence expressions evaluate a series of expressions in order. The value of the sequence expression is value of the last expression.

```
(DefMacro Sequential (&rest expressions)
  (if (null expressions)
      ''()
      (let loop ((exprs expressions))
           (let ((expr (first exprs))
                 (others (rest1 exprs)))
             (if (null others)
                 expr
                 `(let ((ignore ,expr))
                    ,(loop others)))))))
```

Example of use:

```
(Sequence
  (Print (Format nil "~&Hi there!  What's your name?  "))
  (read))
```

11.12 Cond Expressions

This macro implements a sequential cond -- each of the clauses is tried in order until one succeeds. This is implemented as a chain of nested if's.

```
(DefMacro Cond (&rest clause-list)
  (if (null clause-list)
      (second default-last)     ;; if no clauses, complain
      (let loop ((clauses clause-list))
        (let ((clause (first clauses))
              (others (rest1 clauses)))
          `(if ,(car clause)
               (then ,(cdr clause))
               (else
                ,(if (cdr others)
                     (loop others)
                     (let ((last (car others)))
                       (if (eq (car last) 'else)
                           last
                           (loop (list last '(else nil))))))))))))
```

Example of use:

```
(let ((price
       (cond ((age :< 5) 0)
             ((age :< 12) 2.00)
             ((or (age :< 18) (age :> 65)) 3.00)
             (else 4.00))))
  price)
```

11.13 Select Expressions

Select does a sequential test to see if the expression evaluated to any of the choices.

```
(DefMacro Select (expression &rest clause-list)
  (let ((value-sym (unused-symbol))
        (default-last '(else (complaint
                              :fell-off-end-of-select
                              :source ,'(select
                                         ,expression
                                         ,@clause-list)
                              :target self))))
    (if (null clause-list)
        (second default-last)       ;; if no clauses, complain
        '(let ((,value-sym ,expression))
           ,(let loop ((clauses clause-list))
              (let ((clause (first clauses))
                    (others (rest1 clauses)))
                (let ((test (if (listp (car clause))
                                '(or ,@(map (lambda (sub-test)
                                              '(eq ,value-sym
                                                   subtest))
                                            (car clause)))
                                '(eq ,value-sym
                                     ,(car clause)))))
                  '(if ,test
                       (then ,(cdr clause))
                       (else
                         ,(if (cdr others)
                              (loop others)
                              (let ((last (car others)))
                                (if (eq (car last) 'else)
                                    last
                                    (loop (list last
                                                default-last
                                                ))))))))))))))
```

Example of use:

```
(select (parse input)
   ('red :red)
   (('green 'blue) :green-or-blue)
   (('yellow 'orange) :yellow-or-orange)
   (else :invalid-selection))
```

11.14 Let* Expressions

Let* is like Let except the arms are evaluated sequentially rather than concurrently, so each binding is available to the arms after it. This is just syntactic sugar for a set of nested let statements, which this macro produces.

```
(DefMacro Let* (bindings &rest body)
  (labels ((recurse (bindings body)
             (if (< (length bindings) 2)
               '(let ,bindings ,@body)
               '(let ,(list (car bindings))
                  ,(recurse (cdr bindings) body)))))
    (recurse bindings body)))
```

11.15 LetRec Expressions

This macro implements a recursive let, where the identifiers to be bound may be used in the expressions to which they will be bound. It is implemented using futures to represent the values before they are known.

```
(DefMacro LetRec (bindings &body body)
   `(let ,(mapcar (lambda (binding)
                    `(,(first binding) (future)))
                  bindings)
      ,@(map (lambda (binding)
               `(replace ,(first binding)
                         forwarding-script ,(second binding)))
             bindings)
      ,@body))
```

Example of use:

```
(letrec ((husband (make-person :spouse wife))
         (wife (make-person :spouse husband)))
  (marry wife husband))
```

11.16 LabeledLet Expressions

This macro implements a labeled let which allows the concise introduction of a function for recursion purposes; it is often used when the first call requires initial values or the function needs to be created with some call dependent free references, so that it is inappropriate to make the function a top level function. It is inspired by the labeled let available in Scheme. This macro uses the LetRec and Lambda macros we defined above.

```
(DefMacro LabeledLet (label bindings &body body)
  (let ((vars (mapcar #'first bindings))
        (exprs (mapcar #'second bindings)))
    `(letrec ((,label (lambda ,vars ,@body)))
       (,label ,@exprs))))
```

Example of use:

```
(DefFunction factorial (n)
 (LabeledLet rangeproduct ((low 1) (high n))
   (cond ((:= high low) low)
         ((:= high (:+ low 1)) (:* high low))
         (else (let ((average (:/ (:+ high low) 2)))
                 (:* (rangeproduct low average)
                     (rangeproduct (average :+ 1) high)))))))
```

Appendix B

Acore Grammar

B.1 Top Level Form

<Acore-Top-Level-Form> =
    <definition> || <command>

B.2 Definitions

<definition> =
    <module-definition> ||
    <macro-definition> ||
    <name-definition> || <constant-definition>

<module-definition> =
    (**DefModule** (<symbol>*)
        <Acore-Top-Level-Form>*)
Note: The <symbol>*'s in the list must be defined in the body of the module so they can be exported.*

<macro-definition> =
    (**Def-Acore-Macro** <symbol> (<symbol>*)
        <body-expression>)
    ||
    (**DefExpander** <symbol> (<symbol> <symbol> <symbol>)
        <body-expression>)

<name-definition> =
    (**DefName** <symbol> <expression>)

<constant-definition> =
    (**DefEquate** <symbol> <expression>)

B.3 Commands and Expressions

<command> =
    <expression> ||
    <let-command> ||
    <if-command> ||
    <request-command> || <reply-command> || <complain-command> ||
    <ready-command> || <replace-command> ||
    <let-except-command> ||
    <with-sponsor-command>

<expression> =
    <identifier> || <constant> ||
    <create-expression> || <script-expression> ||
    <ask-expression> || <call-expression> ||
    <let-expression> ||
    <if-expression> ||
    <future-expression> || <delay-expression> || <race-expression> ||
    <values-expression> || <complaint-expression> ||
    <let-except-expression>
    <boolean-expression> ||
    <with-sponsor-expression> ||
    <lisp-expression>

<body-expression> = <command>* <expression>

B.4 Actor Creation

<create-expression> =
    (**create** <expression> <expression>*)

<script-expression> =
    (**Script** (<symbol>*)
        *[([**:machine-dependent**] [**:actor-types** (<actor-type>*)])]*
        <communication-handler>*)

<communication-handler> =
    (<request-pattern>
     <body-expression>)
    ||
    (**Is-Request** <request-pattern>
        <command>*)
    ||
    (**Is-Reply** <reply-pattern>
        <command>*)
    ||
    (**Is-Complaint** <complaint-pattern>
        <command>*)

<request-pattern> =
    (<keyword> (<communication-binding>*)
      *[**:self** <symbol>]*
      *[**:sponsor** <symbol>]*
      *[**:customer** <symbol>]*
      *[**:reply-keyword** <symbol>]*
      *[**:unserialized** || **:serialized**]*)

<reply-pattern> =
    (<keyword> (<communication-binding>*)
      *[**:self** <symbol>]*
      *[**:unserialized** || **:serialized**]*)

<complaint-pattern> =
    (<keyword> (<communication-binding>+)
      [:self <symbol>*]*
      [:unserialized || :serialized*]*)

<communication-binding> =
    <symbol> <communication-binding> ||
    &key <communication-binding> ||
    &rest <communication-binding> ||
    &optional <communication-binding> ||
    &allow-other-keys <communication-binding> || ε

<identifier> = <symbol> *which is bound in the enclosing scope*
<constant> = <numeral> || <keyword> || <quoted-list-form> ...

<quoted-list-form> = '<list-form> || (quote <list-form>)
<list-form> = <symbol> || <constant> || (<list-form>*)

B.5 Boolean Expressions

<boolean-expression> =
    <identity-expression> || <not-expression> ||
    <and-expression> || <or-expression>

<identity-expression> = (== <expression> <expression>)
<not-expression> = (not <expression>)
<and-expression> = (and <expression>*)
<or-expression> = (or <expression>*)

B.6 Message Passing

<ask-expression> =
    (<expression> <expression> <expression>*)
[This is interpreted as (<:keyword> <target> <args>)]*

<call-expression> =
    (<expression> <expression>*)
[This is interpreted as (<target> <args>) and is equivalent to (:do <target> <args>*)]*

<request-command> =
    (request <expression> (<expression> <expression>*)
        <expression> <expression> <expression>) ||
    (request* <expression> (<expression> <expression>)
        <expression> <expression> <expression>)

<reply-command> =
    (reply-to <expression> (<expression> <expression>*)) ||
    (reply-to* <expression> (<expression> <expression>))

<complain-command> =
(**complain-to** <expression> (<expression> <expression> <expression>*)) ||
(**complain-to*** <expression> (<expression> <expression> <expression>))

<ready-command> =
(**ready** {(<keyword-symbol> <expression>)}*)

<become-command> =
(**become** <expression> <expression>*)

<replace-command> =
(**replace** <expression> <expression> <expression>*)

B.7 Subexpression Evaluation

<future-expression> =
(**future** <expression>) || (**future**)

<delay-expression> =
(**delay** <expression>)

<race-expression> =
(**race** <expression>*)

B.8 Returning Multiple Values

<values-expression> = (**values** <expression>*)

B.9 Binding Intermediate Values

<let-command> =
(**let** ({((<communication-binding>+) <expression>)}*)
<command>*) ||
(**let** ({(<symbol> <expression>)}*)
<command>*)

<let-expression> =
(**let** ({((<communication-binding>+) <expression>)}*)
<body-expression>) ||
(**let** ({(<symbol> <expression>)}*)
<body-expression>)

B.10 Control Flow

<if-command> =
    (if <expression>
        (then <command>*)
        (else <command>*))

<if-expression> =
    (if <expression>
        (then <body-expression>)
        (else <body-expression>))

B.11 Sponsor Control

<with-sponsor-command> =
    (with-sponsor <expression>
        <command>*)

<with-sponsor-expression> =
    (with-sponsor <expression>
        <body-expression>)

B.12 Complaint Generation and Trapping

<complaint-expression> =
    (complaint <expression> <expression>*)

<let-except-command> =
    (let-except (*{*((<communication-binding>+) <expression>)*}**)
            (except-when
             { ((<keyword> <communication-binding>*)
                 <command>*) *}*)
        <command>*) ||
    (let-except (*{*(<symbol> <expression>)*}**)
            (except-when
             { ((<keyword> <communication-binding>*)
                 <command>*) *}*)
        <command>*)

<let-except-expression> =
    (let-except (*{*((<communication-binding>+) <expression>)*}**)
            (except-when
             { ((<keyword> <communication-binding>*)
                 <body-expression>) *}*)
        <body-expression>) ||
    (let-except (*{*(<symbol> <expression>)*}**)
            (except-when
             { ((<keyword> <communication-binding>*)
                 <body-expression>) *}*)
        <body-expression>)

B.13 Interfacing to Lisp

<lisp-expression> =
    #L(<lisp-function> <expression>*) || #I(<lisp-function> <expression>*)

Index

Appendix D

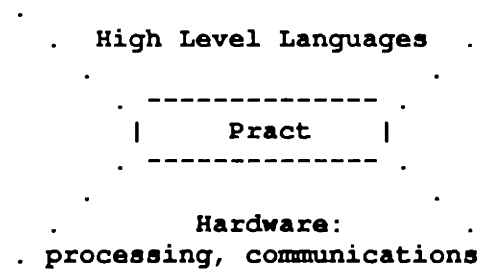
Pract: A Primitive Actor Language Reference Manual

Table of Contents

Chapter One

Introduction

Pract is a primitive actor language designed to serve as an interface between the compiler writer and the computer architect for actor languages. In this sense it serves a function similar to that of assembly languages or p-codes. Systems may be built by building high level actor language compilers which produce Pract code, and simulators, virtual machines, or computer architectures can be built for running Pract. Like assembly language, Pract code itself is not designed to be suitable for direct interpretation, but provides a human readable intermediate form which may then be compiled into implementation specific forms for execution.



Pract was designed to divide the task of implementing an actor computing system into two parts: that of designing and implementing the higher level languages, and that of creating an architecture (simulator, virtual machine, or hardware) to perform actor computations. Pract is the constant interface between these two developing, evolving parts. Pract is based on the Actor model of computation described by Gul Agha in *Actors: A Model of Concurrent Computation in Distributed Systems.*[1] The job of the high level language implementor becomes that of generating the system of continuation behaviors which will control a community of Pract actors in the ways described by the higher level language. (See the appendix of Pract examples to see how one higher level actor language is compiled into Pract.) The task for the architect is to implement the message passing semantics of Pract.

1.1 Guide to this Document

The next chapter provides a brief introduction to the model of computation, as described by Pract programs. It does include a few details specific to Pract, so people familiar with the actor model of computation should at least skim it, paying particular attention to how identity and decision making is handled in Pract. The following chapters are a reference manual for Pract. The first appendix shows some examples of Pract code; I urge anyone trying to understand Pract to study these examples carefully.

[1]MIT AI TR-844, June 1985, soon to be published by MIT Press

A grammar in the second appendix is provided to help clear up any syntactical ambiguities.

Chapter Two

The Actor Model of Computation

The actor model of computation is based on the idea that computation can be performed by sending *messages* between dynamic objects called *actors*. Actors are somewhat like objects in Simula and Smalltalk, except actors are designed to communicate and process messages in a concurrent rather than a sequential environment. In order to make the *guarantee of delivery* of messages possible and to maximize the apparent concurrency available, an actor's behavior in response to receiving a message is limited to the following actions: an actor may make simple decisions, create new actors, send new messages to actors it knows, and specify the behavior which will process its next message. In particular, actors may not make assignments to variables, since this obscures concurrency. Nor may they loop, for two reasons: an actor in a possibly infinite loop cannot receive another message, and strict iteration requires assignment. Therefore all control structures are implemented as patterns of passing messages, so (for example) iterative loops are implemented with tail recursive message passing. Instead of assignment, all actor languages accomodate change through the mechanism of specifying a replacement behavior to handle the next message. Actors are like other object oriented languages in one respect: everything in the actor model is an actor, so all computation is done through message passing.

2.1 The Structure and Behavior of an Actor

An actor has several parts which define its behavior. An actor's *acquaintances* are the other actors it knows about before receiving a message. The acquaintances store the state of an actor; an actor changes its behavior by changing its acquaintances. All actors have at least one acquaintance, its *script*, which is the "program" which tells it what to do when it receives a message. Many actors may share the same script; they will have the same general behavior.

The behavior of an actor is how it reponds to messages. As stated above, in response to a message an actor may only make simple decisions, create new actors, send new communications, and specify a replacement behavior. Each of these may be carried out only using the local information available to the actor -- the identity of the actors known. An actor may "know" (have a reference to) another actor only because the other actor is a constant in the script, is an acquaintance, was part of the incoming message, or was just created.

- **Decisions:** Since actors are designed to run concurrently, actors can only make decisions based on the local information available to them. In particular, this means they cannot make decisions based on the state of other actors, so they are limited to making decisions based on the identity of actors.
- **Creations:** Since actors communicate only by message passing and do not have access to each other's internal state, the state of a new actor must be completely specified when it is created. This means the script and all other acquaintances of the new actor must be known

to the creating actor.

- **Communications:** Actors may send messages to other actors it knows, and the messages may only contain references to actors it knows.
- **Behavior Replacement:** An actor must specify a replacement behavior to process the next message. The behavior may be identical to the current behavior, in which case there is no change in behavior. The actor may keep the same general behavior but change some acquaintances. Or the actor may keep only its identity and take on a completely new behavior by changing its script and its acquaintances.

A behavior is divided into different handlers, one for each message it can process. Each handler may be described as either *serialized* or *unserialized.* A handler is *serialized* if it may change the actor's state; it is *unserialized* if it never changes the behavior. These terms are also applied to actors as a whole, so an actor which cannot change its state (all its handlers are unserialized) is called an unserialized actor, while other actors are serialized actors. Thus, for example, numbers (e.g. 1, 2, 3.14159, etc.) and functions (e.g. factorial, sine, etc.) are all unserialized actors, while actors which change state (e.g. bank accounts) are serialized actors.

2.2 Communications Between Actors

The message delivery system of the actor model is similar in character to a mail system. Communication between actors is *asynchronous*; when one actor sends another a message, the sender does not wait for the receiver to get the mail, but may move on to process its next message. Communication is also **buffered**; if an actor is not ready to receive a message when it arrives, it is queued in its mailbox.

To send a message to another actor, an actor must know the receiver's *mail address.* Therefore all references to actors are mail addresses: actors remember the mail addresses of their acquaintances, and messages sent between actors contain only mail addresses. An actor's mail address is unique: every mail address belongs only to one actor, and each actor has only one mail address. Thus actors can make decisions based on the identities of actors by comparing whether their mail addresses are identical. However, the fact that two mail addresses are different does **not** imply that sending a message to each one must have a different effect; the behavior of one of the actors (if it is a forwarding actor) may be to forward all its messages to the other actor. For example, any actor which behaves as the actor **nil**, including any actor which forwards all its messages to the actor **nil**, will respond to messages in the same way as **nil** but does not have the same identity as **nil** and will fail any condition which compares its identity to **nil.**

Messages are handled by the mail system in packets called *tasks*, which you can think of as envelopes. A task contains the message to be transmitted and also holds the mail address of the recipient, sometimes known as the *target* of the task. The execution of a task is called an *event*; it is the acceptance of a communication by an actor. An event is atomic; it cannot be interrupted by another communication

or any other action in the system. All of the operations performed by an actor in response to a message are done at once without interruption; since the actor cannot loop, it is guaranteed to terminate.

Communications are divided into two general categories. A *request* communication is analagous to a function or procedure call in traditional computer languages. The *response* can be one of two types, either a *reply* communication or a *complaint* communication. Replies are like normal procedure or function call returns, while complaints signal exceptional returns. A request task and its responding task make a pair called a *transaction*.

Each of these communications contains a *keyword* which designates which of the actor's handlers should be used to process the message. For example, a request to a bank account for its balance would have the :balance keyword and be processed by the handler for :balance requests, while a request for a withdrawal would have the :withdrawal keyword and would be processed by the handler for :withdrawal requests. The message may have zero or more additional arguments; these serve the purpose of parameters to a procedure call or returned values of a function in traditional computer languages.

In addition to the keyword and any other arguments, request communications have some additional parameters. A request must contain the mail address of the *customer*, the actor to whom the response should be sent. It must also contain the *reply keyword* which will be used in a normal reply to designate which of the customer's reply handlers should be used to process the reply. Finally, the request must designate a *sponsor* which will oversee the transaction (sponsors will be explained more fully later).

When an actor receives a message, the system must find the handler with the right type (request, reply, complaint) and keyword to process the message. If it cannot find a handler with the correct keyword, then it selects the *otherwise handler* to process the message.[2] Default handlers are provided for some messages which all actors need to be able to process, such as producing a string for printed output. Default otherwise handlers are also provided. For example, the search for a request handler may be visualized as a top to bottom search through the following:

[2]A more complicated search for a handler (such as one using inheritance as in Smalltalk) has been avoided because it doesn't fit in well with the concurrent running environment where actors respond to messages based only on local state information. Instead we are investigating sharing behavior through delegation, implemented with message passing.

Handlers from Script
Request Handlers
keyword1 Script handler 1
keyword2 Script handler 2
. .
. .

Handlers from System
Default Request Handlers
keyword1 Default handler 1
keyword2 Default handler 2
. .
. .

Script Request Otherwise Handler

Default Request Otherwise Handler

Otherwise handlers are important for handling spurious, unrecognized messages. The reply otherwise handler is separated from the complaint otherwise handler because an unrecognized reply generally indicates a programming error and should generate a complaint, while a complaint otherwise handler may be used to trap errors and retry, or may just relay the complaint.

2.3 Resource Management with Sponsors

The use of resources by the many branches of an actor computation is controlled through *sponsors*. Each transaction must have a sponsor actor which "pays for" executing the transaction. When a request is executed, the system may request a resource of "ticks" from its sponsor. If the sponsor has ticks available, it supplies some ticks and the task is executed. If there are any ticks left over afterward, the remaining ticks are returned to the sponsor. When the sponsor runs out of resources, it may ask its parent sponsor for more resources. Depending on the state of the computation, the parent sponsor may or may not grant additional resources, and may take time making its decision or waiting for additional information. Once a decision is made, if additional resources are granted, the computation may continue; otherwise a sponsor may decide a subcomputation is no longer needed and force any pending transactions to abort.

For example, if a parallel search is being performed, the parent sponsor can give each branch of the search a small number of ticks. Each branch must then periodically report back asking for more ticks. If one branch finds an answer, then the other branches can be told to abort the next time they report back for more ticks again, conserving computational resources.

Ticks for a transaction are charged at the time of the request to make sure the transaction can be aborted; otherwise the problem arises that a transaction stopped because of lack of funds can't be aborted cleanly because of lack of funds to send the complaint messages.

The requirement that each transaction be sponsored also solves the problem of controlling runaway

computations without stopping all computations running in the system. The runaway computations can simply be denied further resources, so when they run out of resources they must abort.

2.4 Continuations

Actor computations make wide use of short-lived actors as *continuations*. When an actor must make subtransactions before the response to a request can be determined, the actor (sometimes called the *leading actor*) creates a continuation actor to serve as the customer for the subtransaction. Any actors which will be needed to continue computing the response once the subtransaction is complete are stored as acquaintances of the continuation. This probably includes the customer to whom the final response should be sent, the keyword with which a reply should be sent, and the sponsor which will oversee any further subtransactions. It may also include parts of the original request, acquaintances of the leading actor, or the leading actor itself. Since sequential transactions are performed by sequences of continuations, intermediate results may also be acquaintances. In this sense the set of acquaintances of a continuation actor are like a stack frame, and the behavior of the continuation is like the return address in a conventional machine. Since customers are generally continuation actors, these "frames" are linked together through the customer links; these chains resemble stacks except that they may be distributed throughout the machine and the "stack" may fork due to parallel subtransactions. Unlike some stacks models, however, continuations may not access other stack frames.

If the leading actor can specify its replacement behavior without any intermediate results (for example, if the behavior stays the same), then it is free to accept its next communication immediately following the request event and the continuation will take care of completing the first transaction. Since the continuations are created dynamically for each transaction, many transactions to the same actor may be computed in parallel, or a transaction may recursively make requests to the same actor.

If the replacement behavior of the actor which received the request depends upon the intermediate results, the actor specifies the *insensitive behavior* as its replacement behavior. While the actor is insensitive, it accepts no messages except an *update* or *replace* communication sent by a continuation at the point in the computation when the replacement behavior is determined. The update or replace communication specifies the replacement behavior, and after it is received the actor may accept further communications. Update communications are used to make incremental changes to the behavior, e.g. change an acquaintance or two, while replace communications are used to make complete changes of the behavior. Update is also used when the continuation decides there should be no change, e.g. when some subtransaction is aborted due to an error.

The factorial example in the appendix shows how continuations are used to implement expression evaluation. The bank account example shows how continuations use the update communication to designate a replacement behavior for the leading actor once it has been computed. It also shows how the continuations make sure leading account is unlocked when some transaction is aborted. Both examples

show how continuations use update to designate their own replacement behaviors between sequential transactions.

Chapter Three

Top Level Pract Forms

3.1 DefModule

(DefModule *[module-name] (exported names...)* *TOP LEVEL FORM*
top level forms...)

DefModule is a form for restricting the scope of names defined by **defpract** and **defname**. By default, the names defined inside of a module are visible only in the module, and are not available outside the module. If a name defined inside a module is listed in the list of *exported names*, then the name will be visible in the environment enclosing the module as well. *Module name* is optional; in the future it may be used as a name for the module when composing modules.

Example:

```
(defmodule (square) (*)
  (defpract square-script () (*)
    (is-request (:do n)
      (request * (:do n n) sponsor customer reply-keyword)))
  (defname square (create square-script)))
```

3.2 DefName

(DefName *name pract-expression*) *TOP LEVEL FORM*

DefName enters *name* into the Pract loader table and defines a special forwarding actor to the value of pract-expression. If *name* is new or was not previously defined with **DefName**, a forwarding actor which forwards to the value of *pract expression* is created and bound to *name*. If *name* is being redefined, the forwarding actor is sent a message to redirect any future messages to the value of *pract expression*. Thus using **DefName** allows you to compile programs incrementally (cf. **DefEquate**).

Note that the *script* used in **create** expressions must be a script actor and not a forwarding actor to the script, so it is usually wrong to name a script with **DefName**; use **DefEquate** instead.

When the loader comes across an undefined name, it will issue a warning and **DefName** the name to an "undefined" actor which generates an error if it receives a name. The loader expects that the name will be **DefName**'d later.

Example:

```
(DefName my-account (create checking-account-script 1000))
(DefName spouses-account my-account)
(DefName my-account (create savings-account-script 20000))
```

After loading the above forms, both `my-account` and `spouses-account` refer to a

forwarding actor to a savings account.

3.3 DefEquate

(DefEquate *name pract-expression*) *TOP LEVEL FORM*

DefEquate enters *name* into the Pract loader table and binds the value of *pract expression* directly to *name*. If *name* was already in the loader table, previous uses of *name* will not get the new definition. Therefore, when a name defined with **DefEquate** is redefined, all code which uses that name should be reloaded.

DefEquate is usually the right thing to use for scripts, since the script argument to a **create** expression must be a script actor and not a forwarding actor to a script (cf. **DefName**).

Example:

```
(DefEquate my-account (create checking-account-script 1000))
(DefEquate spouses-account my-account)
(DefEquate my-account (create savings-account-script 20000))
```

After loading the above forms, `my-account` refers directly to a savings account, while `spouses-account` refers directly to a checking account.

3.4 DefPract

(DefPract *name* (*acquaintances...*) (*imported names...*) *TOP LEVEL FORM*
{ ([:machine-dependent*] [*:actor-types (*types...*)]) }**
handlers...)

DefPract is a form for defining behaviors, i.e. script actors. It is equivalent to defining a constant to be the value of a script expression. For example, the following two forms are equivalent:

```
(DefPract cartesian-point-script (x y) (cartesian-to-polar)
   (is-request ((:polarize) :unserialized)
    (request cartesian-to-polar (:do x y)
     sponsor customer reply-keyword)))

(DefEquate cartesian-point-script
  (script
    ((x y) (cartesian-to-polar)
     (:actor-types cartesian-point-script)
     (is-request ((:polarize) :unserialized)
      (request cartesian-to-polar (:do x y)
       sponsor customer reply-keyword)))
    cartesian-to-polar))
```

Chapter Four

Defining Actors

Every actor has an acquaintance called its *script* which defines its general behavior; it specifies how the actor will respond to messages in terms of the actor's acquaintances. Scripts are actors which are created by **script** expressions; once a script is defined, many actors with the same general behavior represented by the script may be constructed using **create** expressions.

4.1 Script

(Script *EXPRESSION*
 ((*acquaintances...*) (*constant names...*)
  { ([:machine-dependent*]* *[*:actor-types (*types...*)*]*) *}**
  handlers...)
 constant actors...

Script is a form for defining behaviors, i.e. script actors. The script actor defined is returned as the result of the expression. All actors which have this script will have the general behavior defined by this script; in particular, they will respond to messages in the general way defined by the *handlers*. Each actor with this script may have different *acquaintances*, so the specific response to a message may depend upon how the handler reacts to the identity of the individual actor's acquaintances. Handlers may also refer to the actors referenced by the *constant names*, but these actors will be identical for all actors with this script. The acquaintances are used to refer to those actors which will be different for different instantiations of the behavior, such as the balance of a bank account, and may or may not change over time. The constant names refer to actors which are the same for all instantiations of the behavior, such as the sine function actor. The Pract expressions *constant actors* specifies the actors to which the constant names will be bound.

The **:machine-dependent** option, if present, specifies that instantiations of this behavior depend upon the the machine on which they are created and should not be migrated to other machines. This is to be used whenever an actor depends upon the hardware of a particular machine in some way; it is used mostly for actors which represent input/output facilities.

The types of a behavior are simply a list of symbols which records static type information. The first symbol on this list is used by the default actor print handler.

Each handler may take any of the following forms:

(Is-Request (*:keyword args...*)
    commands...)
(Is-Reply (*:keyword args...*)
    commands...)
(Is-Complaint (*:error-keyword reply-keyword args...*)
    commands...)

The handlers contain patterns for messages and the commands to perform when a message is received which matches the pattern. Only the keyword of the message is actually matched with the pattern; the rest of the message is bound to the variables in the argument list of the pattern. Thus in the complaint handler, the *:error-keyword* is matched literally, while *reply-keyword* and any other *args* are bound to the incoming parts of the complaint, namely the reply-keyword of the transaction and any other parameters of the complaint.

At most one request handler, one reply handler, and one complaint handler may be an *otherwise handler*. An otherwise handler matches any message if no other handler matches it.[3] An otherwise handler has a normal symbol instead of a keyword in the the keyword postion of the pattern; the incoming keyword is bound to the symbol. For example, the following is a request otherwise handler:

```
(is-request (keyword &rest args)
   (complain-to customer (:unrecognized-message reply-keyword
                                        :selector keyword
                                        :target self
                                        :customer customer)))
```

By default, each handler is serialized, but if the **:unserialized** option is specified, the handler will be unserialized. A serialized handler locks the actor when it receives a message; the actor will stay locked until an *update* or *replace* message is received. An unserialized handler leaves the actor unlocked when it is through.[4] For documentation purposes, you may also specify the **:serialized** option in a serialized handler.

By default, in request handlers, reply handlers, and complaint handlers, the actor which is receiving the message is bound to **self**. In request handlers only, the sponsor, customer, and reply keyword are bound to the identifiers **sponsor**, **customer**, and **reply-keyword** respectively. To override any of these defaults and provide another symbol to be bound, the following forms may be used (replacing the italicized symbol with your own identifier):

---

[3]See the section on messages for a full explanation of how a handler is found.

[4]An implementation may find it necessary to lock the actor momentarily to make sure the handler gets a consistent set of acquaintances.

```
(Is-Request ((:keyword args...)
                                :self self :sponsor sponsor
                                :customer customer :reply-keyword reply-keyword
                                [:unserialized || :serialized])
        commands...)
(Is-Reply ((:keyword args...) :self self
        [:unserialized || :serialized])
        commands...)
(Is-Complaint ((:keyword reply-keyword args...) :self self
        [:unserialized || :serialized])
        commands...)
```

4.2 Create

(Create *script acqs...*) *EXPRESSION*

Create is the primitive for creating a new actor. The expression *script* designates the actor which will be the script of the new actor. The *acqs...* designate the actor's acquaintances in order. **Create** creates a new actor with *script* as its script and *acqs* as its acquaintances.

Example:

```
(Create account-script 100)
#<ACCOUNT-SCRIPT Actor 5055948>
```

Create is a low level primitive, and does not do any checking. In particular, it does not check whether *script* is a script actor, nor does it check whether the right number of acquaintances are provided. **Create** should not be directly available in a higher level language, but should be replaced with functions which use create, and do the required checking. (See the bank account example in the appendix for an example.)

Chapter Five

Pract Commands

5.1 Request

(Request *target* (*:keyword args...*) *COMMAND*
sponsor customer [*:reply-keyword*])

The **Request** command sends a request message to the actor designated by *target*. The primary content of the message is the *:keyword* and *args*. The transaction will be sponsored by *sponsor*. The response to this request, be it a reply or a complaint, will be sent to *customer*. If it is a normal reply it will be sent with the keyword *reply-keyword*, which defaults to **:value**.

5.2 Reply-To

(Reply-To *target* (*:keyword args...*)) *COMMAND*

The **Reply-To** command sends a reply message to the actor designated by *target*. The primary content of the message is the keyword and any arguments which may be result values to be returned. If this is a normal reply, *:keyword* should probably be the **reply-keyword** which came with the corresponding request.

5.3 Complain-To

(Complain-To *target* (*:error-keyword reply-keywordargs...*)) *COMMAND*

The **Complain-To** command sends a complaint message to the actor designated by *target*. The *:error-keyword* identifies the type of complaint, such as **:unrecognized-request** or **:machine-error-trap**; some types are described in the section about complaints. The *reply-keyword* identifies the transaction which caused the complaint, so when several responses come back to a joining customer, the customer can tell which one(s) complained. Often complaint messages follow the convention that further arguments are preceded by keywords identifying parts of the message, such as the **:target** and **:customer** involved. See the section on complaints for further details.

```
(is-request ((keyword &rest args) :unserialized)
  (complain-to customer (:unrecognized-message reply-keyword
                                    :target self
                                    :selector keyword
                                    :sponsor sponsor
                                    :customer customer)))
```

5.4 Update

(Update *target*) *COMMAND*
(Update *target* (*acq# new-value*) (*acq# new-value*) ...)

The **Update** command sends an update message to the actor designated by *target*. If the actor is locked, this will update the actor's acquaintances indicated and unlock the actor. If the actor is not locked, the system should generate a runtime error. The acquaintances are numbered from 1 in the the order given in the acquaintance list of the actor. Acquaintance 0 is the script of the actor. Thus, a very simple actor can be programmed as follows:

```
(DefPract Memory-Cell (cell) ()
  (is-request (:read)
     (reply-to customer (reply-keyword cell)))
  (is-request (:write new-value)
     (update self (1 new-value))
     (reply-to customer (:ack))))
```

A higher level language should restrict the ability to update so it is available only within a higher level actor's script, i.e. only an actor or its continuations may issue updates to the actor, thereby guaranteeing it must have been locked.

Update is provided to make simple changes to an actor's behavior -- those which do not change every acquaintance or the number of acquaintances. Common changes such as updating the balance of a bank account can be implemented efficiently by using update rather than replace.

[Many, if not most, updates will be performed by continuations rather than the original actor which received the message, but the continuations' scripts do not have direct knowledge of the acquaintance names of the original actor, so the acquaintances are referred to by numbered position rather than by name. Pract is intended as a target language for higher level compilers, so keeping the numbers straight shouldn't be a problem.]

5.5 Replace

(Replace *target script acq1 acq2...*) *COMMAND*

The **Replace** command sends a replace message to the actor designated by *target*. If the actor is locked, this will replace the actor's entire state with the state represented by the new script and acquaintances, i.e. the actor will become an actor which has *script* as its script and *acq1*, *acq2*, ... as its only acquaintances.

Replace is provided to make complete changes in an actor's behavior -- those which change every acquaintances and/or change the number of acquaintances. Note that it doesn't make sense to change the number of acquaintances without changing the script. Thus part of the scripts for a future and the forwarding actor it becomes might look as follows:

```
(DefPract Future-Script (waiting-room) ()
   . . .
   (is-reply (:value result)
      (replace self Forwarding-Script result))
   . . .)

(DefPract Forwarding-Script (forward-to) ()
   (is-request (keyword &rest args)
     (request* forward-to (keyword args)
        sponsor customer reply-keyword))
   . . .)
```

5.6 Let

(Let ((*identifier expression*) (*id expr*) ...) *commands...*) *COMMAND*

The **Let** command allows you to bind a name to a Pract expression and use the name in the commands of its body. This is used primarily for binding a name to a newly created actor so it may be used in several commands, e.g. as the customer to several requests:

```
...(let ((continuation (create cont-1 ...)))
     (request x (:do) s continuation :x)
     (request y (:do) s continuation :y)
     (request z (:do) s continuation :z))
```

When there are several bindings taking place, all the *expressions* are evaluated first, then the names are bound to the expressions. A sequential binding can be performed by nesting Let expressions.

Directly creating a set of recursively referent actors is not possible at this time; it must be accomplished through message passing after the actors are made.

5.7 If

(If *expression* (Then *commands*) (Else *commands*)) *COMMAND*

The **If** command is the conditional command of Pract. Conditional testing is based whether or not the Pract expression *expression* evaluates to a distinguished *false* value or not; in Lisp implementations this value is typically **nil**. Note that an actor which behaves the same as **nil** is not the same as **nil**; in particular, an actor which forwards all its messages to nil is not equivalent to **nil**.

If *expression* does not evaluate directly to **nil**, the commands of the **Then** clause are performed. If *expression* does evaluate directly to **nil**, the commands of the **Else** clause are performed.

5.8 Lisp-Request

(Lisp-Request (*function args...*) *COMMAND*
sponsor customer reply-keyword)

The **Lisp-Request** command is the interface from Pract to a Lisp system running on a particular machine. It is intended to be used to interface Pract actor systems with software systems written in Lisp, such as window systems for user interfacing, file systems for storage, etc. An actor which makes Lisp-requests serves as the interface to the Lisp system from the Pract actor world. It must be **machine dependent** so it won't be migrated to another machine where the instance of the software system or the hardware -- or perhaps even Lisp -- is not available.

When a Lisp-request is made, the Pract implementation forks another process in which to run the request. This guarantees that if the process hangs waiting for an event or goes into a loop, mesage passing may still continue in parallel. If the Lisp request completes successfully, the value returned will be sent to the customer with the reply keyword. If the Lisp request generates an error, a complaint will be sent to the customer describing the error. In both cases the process is terminated upon completion.

5.9 System-Request

(System-Request *target* (*:keyword args...*) *COMMAND*
sponsor customer [*:reply-keyword*])

System request is a command used by system debugging software which issues a special type of request message called a *system request*. System requests are intended only for systems and debugging software; they should not be present in other code. System requests are handled by a special set of system request handlers maintained by the system; they are the same for all actors. See the section on System Requests for the messages which are currently handled.

5.10 Request*, Reply-to*, Complain-To*, System-Request*

(Request* *target* (*:keyword args-nospread*) *COMMAND*
sponsor customer [*:reply-keyword*])
(Reply-to* *target* (*:keyword args-nospread*))
(Complain-to* *target* (*:keyword reply-keyword args-nospread*))
(System-Request* *target* (*:keyword args-nospread*)
sponsor customer [*:reply-keyword*])

The starred (*) form of these message sending commands is used primarily for forwarding arbitrary messages, usually by otherwise handlers. The parameters to the message are collected into an `&rest` argument which is then passed on as an *args-nospread* with one of these commands. For example, a common type of complaint otherwise handler appears something as follows:

```
...
  ;; This transaction botched; unlock target actor so it can handle
  ;; next message, and relay complaint up the customer chain.
  (Is-Complaint ((unknown-selector reply-keyword &rest args)
                 :Unserialized)
    (Update leading-actor)
    (Complain-To* my-customer
        (unknown-selector reply-keyword args)))
...
```

Chapter Six

Pract Expressions

6.1 Symbols

identifier *EXPRESSION*

Symbols can be used as expressions as long as they are bound in the context where they are used. A symbol is bound in a Pract handler if it is either:

1. an *imported name* declared in the header of the DefPract form,
2. an *acquaintance* declared in the header of the DefPract form,
3. a *message argument* bound from the incoming message,
4. a *message keyword* bound in an otherwise handler,
5. one of *self*, *sponsor*, *customer*, *reply-keyword* unless these default identifiers are overridden (see Script), or
6. a *let variable* bound in a Let command or expression.

6.2 Or, And, Not

(Or *expr expr...*) *EXPRESSION*
(And *expr expr...*) *EXPRESSION*
(Not *expr*) *EXPRESSION*

These expressions are mainly useful as conditions of **If** commands. Like the condition of the **If** command, they interpret their argument expressions as either being the *false* value **nil** or not. Other actors which may behave as **nil** are not equivalent to **nil** in this usage. Given this interpretation, **Or** returns **nil** only if all its parameter expressions evaluate to **nil**; **And** returns **nil** if any of its parameter expressions evaluate to **nil**; and **Not** returns **nil** only if its parameter expression does not evalute to **nil**.

6.3 Identity: ==

(== *expr expr*) *EXPRESSION*

== is the identity predicate for testing whether two expressions refer to the same actor. It is primarily useful as a condition in **If** commands. Like the other conditions, == implements the most primitive and most efficient check possible. At the present it is guaranteed to work on any actor created with **create**; in addition it is guaranteed to work on keywords and other symbols, and on **nil**. Currently it is not guaranteed to work on other built-in actors such as lists or (big)numbers, because these representations may be copied by the underlying implementation.

Example:

```
(DefName my-account (create account-script 1000))
(DefName your-account (create account-script 1000))
(DefName my-spouses-account my-account)
...
(== my-account my-account)
true
(== my-account my-spouses-account)
true
(== my-account nil)
false
(== my-account your-account)
false
```

6.4 Quote (')

(Quote *form*) *EXPRESSION*
'form *EXPRESSION*

Quote is used for expressing literals of symbol and list structure in handler code. The *form* itself is returned as the value; *form* is not evaluated. A quote form can be abbreviated as *'form* as in Lisp; the reader expands the apostrophe into the quote form.

```
(Quote Hi)
HI
'Hi
HI
(Quote (Hi there))
(HI THERE)
'(Bye bye)
(BYE BYE)
```

6.5 Numerals

Numeral *EXPRESSION*

Numbers evaluate to themselves.

```
1632
1632
2.54
2.54
```

6.6 Strings

"*string*" *EXPRESSION*

Strings evaluate to themselves.

```
"Hello again."
"Hello again."
```

6.7 Let

(Let (*{(identifier expression)}*) *EXPRESSION*
{ commands...}
expression)

A Pract let expression evaluates the Pract expressions in the arms and binds them to the identifiers; the value of the let expression is the result of evaluating the body expression in the environment extended by the new bindings. Commands may also be performed in the same environment; they are performed concurrently with the evalutation of the expression. A Pract let expression is much like a Lisp let statement with the restriction that the expressions may only be Pract expressions; thus it will be most useful when the expressions are create or script expressions.

A Pract let expression allows directed acyclic networks of actors to be constructed; in particular, it allows directed acyclic networks of scripts to be built as a single expression which returns the leading actor's script. This is important for the implementation of scripts constructed at runtime, such as those implementing the closure produced by a lambda expression in a higher level language.

6.8 If

(If *condition-expression* *EXPRESSION*
(then *{ then-commands... }*
then-expression)
(else *{ else-commands... }*
else-expression))

A Pract if expression evaluates the *condition-expression*; if the condition succeeds or evaluates to a non-nil actor, the value of the if expression is the value of the *then-expression*; otherwise it is the value of the *else-expression*. Commands may also be performed within the branches; they are performed concurrently with the evaluation of the expression.

Pract if expressions aren't strictly necessary, but they help keep down the size of Pract scripts when dealing with Pract expressions by avoiding the need to duplicate the condition. They also make it much simpler to compile if expressions of similar form from higher level languages into Pract.

6.9 In-Lisp

(In-Lisp *Lisp-expression*) *EXPRESSION*

The *Lisp-expression* is evaluated in the lexical environment of the current handler. This form of Pract expression is very implementation dependent, and its use is strongly discouraged. It is an escape to the implementation for writing systems code. It is intended for use only in machine dependent actors for very quick operations such as looking up the value of a variable, e.g. a status flag on a machine with an I/O device. For operations which may take longer and for general interfacing to Lisp systems, see the **Lisp-request** command.

Chapter Seven

Built-in Actors

Built-in actors are those actors in a system at which the message passing "bottoms out", whose behaviors are implemented in the hardware or by firmware. In a Lisp-based implementation, these will often be many of the datatypes provided by Lisp. For efficient implementation, some of these actors will usually be represented in the form most convenient to the host machine, so they are not given mailboxes. Since they do not have mailboxes, they must be copied from machine to machine; therefore they must be immutable.

Built-in actors respond to a variety of messages, some of which are described below. Many messages are implementations of equivalent Lisp function calls; for a full description of their parameters and the values returned, see a Common Lisp manual. For these messages, the Lisp function calls of the forms:

```
(functionA arg1)
(functionB arg1 arg2)
```

can respectively be translated into request commands:

```
(request arg1 (:functionA) sponsor customer :reply-keyword)
(request arg1 (:functionB arg2) sponsor customer :reply-keyword)
```

If there are no errors, then the value which would have been returned by the function is returned to the customer in a reply which is identical to one sent by the following command:

```
(reply-to customer (reply-keyword value))
```

Currently, all errors trapped by the hardware generate complaints which look something like the one generated by the following command:

```
(complain-to customer (:machine-error-trap reply-keyword
                       :target <target actor>
                       :selector <:keyword of message>
                       :further-arguments (list of other parameters)
                       :customer actor to whom complaint was originally sent
                       :reply-keyword <:reply-keyword of message>
                       :report "string describing the error"))
```

This may be changed so the complaint keyword is more descriptive; the current form is is ok for debugging but is not sufficient for trapping errors with complaint handlers.

In order to handle forwarding actors as arguments in messages implementing binary operations on built-in actors, built-in actors which expect another built-in actor as an argument trap the error and reverse the message. For example, if we sent the message

```
(request 5 (:- #<Forwarding-Script Actor 52353>) s c :k)
```

the 5 would find it could not calculate the result using the identity of the forwarding actor, so it would

forward the reversed message to the forwarding actor. The reversed message would look something like this:

```
(request #<Forwarding-Script Actor 52353> (:-% 5) s c :k)
```

The keyword with a special character ('%' in this example[5]) appended to the keyword ':-' invokes a message handler where the arguments are interpreted in the opposite manner. Thus if the forwarding actor forwarded to a 2, the 2 would then calculate the difference (5-2) and return the result to the customer. This reverse occurs only once; if second target cannot recognize the argument to the reversed message (with the '%'), then a complaint is generated.

7.1 Numbers: Fixnums, Bignums, Single-float

All of these types of numbers are currently supported; in addition, further types (such as Double-float) may be added in the future. They are based on the standards described in Common Lisp. There are too many messages to enumerate them all here; suffice it to say that most 1 and 2 parameter Common Lisp functions on numbers are supported in the fashion described above.

7.2 Symbols, Keyword Symbols

Keyword symbols are used in Pract to select which handler of an actor will be used to receive a message. Therefore a keyword with the same spelling must always represent the same actor; otherwise communcations between actors will break down. SInce the primary feature of a keyword symbol is its identity, there aren't really many interesting messages you can send to it, but some Lisp functions are supported as messages.

7.3 Lists

Identity of lists. Lists are currently represented in Pract directly as Lisp list structure so that non-mutating list operations can be performed on them quickly. This means that lists must be copied between machines, so that == may not work on lists. Lists in Pract are immutable.

Creating lists. Currently the only way to create lists is either through the use of `Quote` or by using functions such as `List` with the `Lisp-Request` or `in-lisp` forms. This will change in the future.

Operations on lists. Lists respond to selector messages corresponding to the Lisp functions such as :car, :cadr, :first, :second, :cdr, etc.

---

[5]Implementations should try to choose a character unlikely to be found in user defined message keywords to prevent conflicts

7.4 Arrays and Strings

Arrays are currently implemented as an immutable datatype; they are currently provided primarily to implement operations on strings. Mutable arrays may become available in the future. Lisp operations are available as messages to arrays as described above; the most important of these is probably `:aref` *subscript*. Creation of arrays is currently accomplished by by using " ... " for strings or by using Lisp functions in conjunction with `Lisp-Request` or `in-lisp` forms. This will change in the future.

Chapter Eight

System Provided Actors

Just as most system provide a set of datatypes and functions for performing operations on the datatypes (and possibly i/o if i/o streams are not a datatype), Pract provides some actors which implement datatypes and operations. This set is currently very small, and may be eliminated -- this kind of functionality really should lie in a higher level language specification.

8.1 Null-customer

This actor is used as a bottomless pit for unwanted replies.

8.2 Futures

A future is an actor which waits for a reply, and becomes a forwarding actor which forwards messages to the actor which is the value of the reply. Any requests which are received by the future are queued up and later forwarded to the reply value when it is know. Futures are used for implementing eager evaluation.

```
(request make-future (:do) sponsor customer :reply-keyword)
(Create future-script nil)
```

both create a future; the latter form is subject to change.

Currently futures do not handle complaints; this is a problem.

8.3 Forwarding Actors

A forwarding actor is an actor which forwards any message it receives to another actor. For example, a future actor changes its behavior to that of a forwarding actor once it receives a reply with the following command:

```
(replace self forwarding-script actor-to-forward-to)
```

8.4 Format

The Format actor implements a primitive form of the Lisp format function. It expects a message of the form:

```
(:do stream format-string &rest args)
```

Stream may be nil, in which case the formatted string is returned; it may be T, in which case output is presented on the user's terminal output stream; or it may be an actor which accepts a message of the form

```
(:do formatted-string)
```

Format is limited in that it does not allow one to specify the printed representation of actors. All it does is to ask each actor in `args` a `:string-for-printing` message, and combines these strings with the format string with the lisp Format function.

Chapter Nine

Default Handlers

There are some messages we would like all actors to be able to handle. Other object oriented systems handle the problem of providing actors with default handlers using a behavior inheritance mechanism or a delegation mechanism, but since the topic is currently under research for actors, no behavior sharing mechanism has been implemented. Instead, default handlers are provided for these messages. A behavior may override the default message handler by providing its own handler for the message. (See section on handlers for a description of how a handler is found for a message.)

9.1 Default Otherwise Request Handler

This handler is called if no other request handler for a message is found. It is unserialized, so the actor is left unlocked. It performs the following:

```
(complain-to customer (:unrecognized-request reply-keyword
                                    :target actor
                                    :customer customer
                                    :sponsor sponsor
                                    :selector selector keyword
                                    :arguments list of other arguments))
```

9.2 Default Otherwise Reply Handler

This handler is called if a customer receives a reply for which it has no reply handler. The reply is turned into a **:machine-error-trap** complaint which is sent to the customer instead.

9.3 Default Otherwise Complaint Handler

This handler is called if no other complaint handler is found for a complaint. It is unserialized, so the actor is left unlocked. At present, it does nothing; you should override it with a default complaint handler which does something useful such as forward the complaint up the customer chain, and if the receiver is a continuation, possibly unlock the leading actor so it can receive further messages.

9.4 :Are-You

(:are-you *type-symbol*) MESSAGE

The default response to the `:are-you` message is to return a non-nil value if the actor is of the type described by *type-symbol*; otherwise it returns nil.

9.5 :Types

(:types) MESSAGE

The default response to the `:types` message is to return the list of types of an actor, i.e. a list of those symbols such that asking the actor `:are-you` with the symbol returns a non-nil value.

9.6 :Script

(:script) MESSAGE

The default response to the `:script` message is to return the actor's script actor, which defines the actor's general behavior.

9.7 :Self

(:self) MESSAGE

The default response to the `:self` message is to return the actor itself. This is used to to find the identity of the actor to which a forward actor forwards its message, i.e. what actor really receives the message.

9.8 :String-for-Printing

(:string-for-printing) MESSAGE

The default response to the `:string-for-printing` message is to return a string which describes the actor. For built-in actors, this string is the same as that produced by the Lisp printer; this currently does not work well for lists if components of the list are not on the same machine. For actors whose behavior is defined with DefPract, the string will look something as follows:

```
"#<ACCOUNT-SCRIPT Actor 55043201>"
```

The name of the actor's script will be used as the name. The actor's UID is used as the number if it has a UID; otherwise a number is generated using a hashing function. This number is displayed to help differentiate several actors with the same script.

Chapter Ten

System Requests

System Requests are special messages used by systems code and especially the debugging systems for communicating with actors. System requests should not be used by normal user code at any time. They are documented here for completeness.

System requests provide the debugger with the special capability to communicate with locked actors. This dangerous capability is necessary to display the state of a computation which has deadlocked or otherwise stopped because of program bugs.

10.1 :Acquaintances

(:acquaintances) SYSTEM MESSAGE

The `:acquaintances` system request returns a list of the actor's current acquaintances in order.

10.2 :Acquaintance Names

(:acquaintance-names) SYSTEM MESSAGE

The `:acquaintance-names` system request returns a list of names of the actor's acquaintances, as specified in its script.

10.3 :Are-You

(:are-you *type-symbol*) SYSTEM MESSAGE

See the `:are-you` default request.

10.4 :Biography

(:biography) SYSTEM MESSAGE

The response to the `:biography` system message is to return the actor's biography. The biography represents a history of recorded messages which the actor received; unrecorded messages are not represented in this history. See the Apiary Memos on Traveler for more information on biographies and recording debugging information.

10.5 :Incoming Queue

(:incoming-queue) SYSTEM MESSAGE

The `:incoming-queue` system message returns the actor's queue of messages waiting to be processed.

10.6 :Script

(:script) SYSTEM MESSAGE

See the `:script` default request.

10.7 :Self

(:self) SYSTEM MESSAGE

See the `:self` default request.

10.8 :String-for-Printing

(:string-for-printing) SYSTEM MESSAGE

See the `:string-for-printing` default request.

10.9 :Types

(:types) SYSTEM MESSAGE

See the `:types` default request.

10.10 :Uid

(:uid) SYSTEM MESSAGE

The `:uid` system message returns the actor's unique identifier, if it has one. The uid is the number used to identify the actor in communications between machines; if no references to the actor have migrated to other machines, the actor may not have a UID, so this will return NIL. This aspect if its behavior may change.

Appendix A

Example Programs

The examples which follow show how a higher level actor language could be compiled into Pract.

A.1 A Simple Recursive Factorial

Factorial is a simple example which shows how expressions and a simple conditional in a high level actor language may be unraveled into a chain of simple Pract behaviors.

Below is a possible high level representation of Factorial. We assume that a *call expression*, which has the form `[target arg1 arg2 ...]`, is a shorthand for an *ask expression*, which has the form `(target :do arg1 arg2 ...)`. The value of an ask expression is the reply returned as a result of sending `target` a request with the given keyword and arguments. We could introduce actors for the individual mathematical functions such as * and ≤ but we have chosen to implement these as direct messages to the actors (numbers) involved.

```
(DefFunction factorial (n)
  (if (:≤ n 1)
      (then 1)
      (else (:* n (factorial (:1- n))))))
```

Notice that in the Pract version, the actor which receives the initial message creates a customer which serves a similar function to a stack frame, storing intermediate state between calls to other actors. This customer takes on different behaviors as each subcalculation is completed, so at each call it represents the continuation of the computation. For clarity, the behaviors of the sequence of continuations are defined by the scripts named `fact-c1`, `fact-c2`, etc.

```
(defmodule (factorial) ()

  (defpract factorial-script () (fact-c1)
    (is-request ((:do n) :unserialized)
      (request n (:≤ 1)
          sponsor
          (create
            fact-c1                          ; script
            n                                ; original argument
            customer                         ; top-customer
            reply-keyword                    ; top-reply-keyword
            sponsor                          ; top-sponsor
            ))))

  (defname factorial (create factorial-script))

  (defpract fact-c1 (n top-customer top-reply-keyword top-sponsor)
```

```
                (fact-c2)
  (is-reply (:value n-less-than-1?)
    (if n-less-than-1?
      (then (ready self)
         (reply-to top-customer (top-reply-keyword 1)))
      (else
         (update self (0 fact-c2))              ; acquaintance 0 is the script
         (request n (:1-) top-sponsor self))))
  (is-complaint (keyword replied-keyword &rest args)
    (ready self)
    (complain-to* top-customer (keyword replied-keyword args))))

(defpract fact-c2 (n top-customer top-reply-keyword top-sponsor)
     (fact-c3 factorial)
  (is-reply (:value n-minus-1)
    (update self (0 fact-c3))
    (request factorial (:do n-minus-1) top-sponsor self))
  (is-complaint (keyword replied-keyword &rest args)
    (ready self)
    (complain-to* top-customer (keyword replied-keyword args))))

(defpract fact-c3 (n top-customer top-reply-keyword top-sponsor) ()
  (is-reply (:value value-minus-one-factorial)
    (ready self)
    (request n (:* value-minus-one-factorial)
        top-sponsor top-customer top-reply-keyword))
  (is-complaint (keyword &rest args)
    (ready self)
    (complain-to* top-customer (keyword args))))

) ; end defmodule
```

A.2 A Parallel Factorial

This example demonstrates how parallel transactions can dispatched simultaneously in Pract, and the replies can be synchronized with the help of reply keywords.

In a higher level language, this might be programed as follows (the `Let` syntax for introducing an auxiliary function is stolen from Scheme). Note that the arms of the `Let` are evaluated in parallel.

```
(DefFunction rk-factorial (n)
 (let rk-rangeproduct ((low 1) (high n))
  (cond ((:= low high)
         low)
        ((:= (:- high low) 1)
         (:* low high))
        (else
         (let ((average (:/ (:+ low high) 2)))
           (let ((successor-of-average (:1+ average)))
             (let ((low-product (rk-rangeproduct low average))
                   (high-product (rk-rangeproduct
                                  successor-of-average high)))
               (:* low-product high-product)))))))))
```

In the Pract version, note that the first request after joining the two parallel transactions is duplicated in two handlers, since it is not known which reply will return last. The redundancy is not excessive since only the first request is duplicated; further continations would be shared. Also note that complaint handlers have been omitted for brevity.

```
(defmodule (rk-factorial) ()
  (defname rk-factorial (create rk-factorial-script))

  (defpract rk-factorial-script () (rk-rangeproduct)
    (is-request ((:do n) :unserialized)
      (request rk-rangeproduct (:do 1 n)
         sponsor customer reply-keyword)))

  (defmodule  (rk-rangeproduct) ()

   (defname rk-rangeproduct (create rk-rangeproduct-script))

   (defpract rk-rangeproduct-script () (rk-rp-1)
    (is-request ((:do low high) :unserialized)
                 (let ((cont (create rk-rp-1
                                     customer reply-keyword sponsor
                                     low high nil)))
                  (request low (:= high) sponsor cont))))

   (defpract rk-rp-1 (top-customer top-reply-keyword top-sponsor
                      low high ignore) (rk-rp-2)
    (is-reply (:value equal?)
     (if equal?
      (then (reply-to top-customer (top-reply-keyword low))
        (ready self))
      (else (update self (0 rk-rp-2))
        (request high (:- low) top-sponsor self)))))

   (defpract rk-rp-2 (top-customer top-reply-keyword top-sponsor
                      low high ignore) (rk-rp-3)
    (is-reply (:value difference)
     (update self (0 rk-rp-3) (6 difference))
     (request difference (:= 1) top-sponsor self)))

   (defpract rk-rp-3 (top-customer top-reply-keyword top-sponsor
```

```
                    low high difference) (rk-rp-4)
  (is-reply (:value consecutive?)
   (if consecutive?
    (then (request low (:* high)
              top-sponsor top-customer top-reply-keyword)
      (ready self))
    (else (update self (0 rk-rp-4))
      (request low (:+ high) top-sponsor self)))))

 (defpract rk-rp-4 (top-customer top-reply-keyword top-sponsor
                    low high ignore) (rk-rp-5)
  (is-reply (:value sum)
   (update self (0 rk-rp-5))
   (request sum (:// 2) top-sponsor self)))

 (defpract rk-rp-5 (top-customer top-reply-keyword top-sponsor
                    low high ignore) (rk-rp-6)
  (is-reply (:value average)
   (update self (0 rk-rp-6) (6 average))
   (request average (:1+) top-sponsor self)))

 ;; The continuation which dispatches the parallel calls
 (defpract rk-rp-6 (top-customer top-reply-keyword top-sponsor
                    low high average) (rk-rangeproduct rk-rp-7)
  (is-reply (:value successor-of-average)
   (update self (0 rk-rp-7) (6 2))
   (request rk-rangeproduct (:do low average)
      top-sponsor self :low-product)
   (request rk-rangeproduct (:do successor-of-average high)
      top-sponsor self :high-product)))

 ;; The continuation which receives the replies from the parallel calls
 (defpract rk-rp-7 (top-customer top-reply-keyword top-sponsor
                    low high nwaiting) ()
  (is-reply (:low-product low-value)
   (if (== nwaiting 2)
    (then (update self (4 low-value) (6 1)))
    (else (ready self)
      (request low-value (:* high)
          top-sponsor top-customer top-reply-keyword))))
  (is-reply (:high-product high-value)
   (if (== nwaiting 2)
    (then (update self (5 high-value) (6 1)))
    (else (ready self)
      (request low (:* high-value)
          top-sponsor top-customer top-reply-keyword)))))
 ) ;; end of rk-rangeproduct module
) ;; end of rk-factorial module
```

A.3 A Simple Bank Account

This example shows how a serialized behavior may be programmed using Pract.

In a higher level language, the bank account might be programmed as follows:

```
(def-module (make-account)
  (DefScript account (balance)
    (is-req (:balance) balance)
    (is-req (:deposit deposit-amount)
      (let ((new-balance (balance :+ deposit-amount)))
        (ready (:balance new-balance))
        ':done))
    (is-req (:withdrawal withdrawal-amount)
      (let ((new-balance (balance :- withdrawal-amount)))
        (if (new-balance :< 0)
          (then
            (ready)
            (complain (:overdraft)))
          (else
            (ready (:balance new-balance))
            withdrawal-amount)))))
   (DefFunction [make-account starting-balance]
     (new account (:balance starting-balance))))
```

In the Pract version, note that the account actor (as opposed to the customer it creates) stays locked until a continuation issues an `update` (except for the trivial `:balance` request which doesn't require a continuation).

```
(defmodule (make-account) ()

  (defmodule  (account) ()

   (defpract account (balance) (account-c1 account-c2)
    (is-request ((:balance) :unserialized)
     (reply-to customer (reply-keyword balance)))
    (is-request (:deposit deposit-amount)
     (let ((continuation (create account-c1
                           self customer reply-keyword)))
      (request balance (:+ deposit-amount) sponsor continuation)))
    (is-request (:withdrawal withdrawal-amount)
     (let ((continuation (create account-c2
                           self withdrawal-amount nil
                           customer reply-keyword sponsor)))
      (request balance (:- withdrawal-amount) sponsor continuation))))

   (defpract account-c1 (account top-customer top-reply-keyword) ()
    (is-reply (:value new-balance)
     (reply-to top-customer (top-reply-keyword ':done))
     (update account (1 new-balance)))
    (is-complaint (complaint-keyword reply-keyword &rest args)
     (udpate account) (update self)
     (complain-to* top-customer
        (complaint-keyword top-reply-keyword args))))
```

```
(defpract account-c2 (account withdrawal-amount balance
                      top-customer top-reply-keyword top-sponsor)
    (account-c3)
 (is-reply (:value new-balance)
  (update self (0 account-c3) (3 new-balance))
  (request new-balance (:< 0) top-sponsor self))
 (is-complaint (complaint-keyword reply-keyword &rest args)
  (update account) (update self)
  (complain-to* top-customer
     (complaint-keyword top-reply-keyword args))))

(defpract account-c3 (account withdrawal-amount new-balance
                      top-customer top-reply-keyword top-sponsor)()
 (is-reply (:value overdraft?)
  (if overdraft?
   (then
     (complain-to top-customer (:overdraft top-reply-keyword))
     (update account))
   (else
     (reply-to top-customer (top-reply-keyword withdrawal-amount))
     (update account (1 new-balance))))
  (update self))
 (is-complaint (complaint-keyword reply-keyword &rest args)
  (update account) (update self)
  (complain-to* top-customer
     (complaint-keyword top-reply-keyword args))))
) ;; end account sub-module

(defmodule (make-account) (account)

 (defpract make-account-script () (account)
  (is-request ((:do starting-balance) :unserialized)
   (reply-to customer
      (reply-keyword (create account starting-balance)))))

 (defname make-account (create make-account-script))
) ;; end make-account sub-module
) ;; end module
```

Appendix B

Pract Grammar

10.11 Top Level Forms

<pract-top-level-form> =
    <name-definition> ||
    <constant-definition> ||
    <send-command> ||
    <script-definition> ||
    <module-definition>

<module-definition> =
    (**defmodule** *[* <loader-identifier> *]*
        (<loader-identifier>*) (<loader-identifier>*)
        <pract-top-level-form>*)
Note: The first (optional) <loader-identifier> *may be used in the future to combine modules as configurations. The* <loader-identifier>*'s in the first list must be defined in the body of the module so they can be exported. The* <loader-identifier>*'s in the second list should be defined in the lexically enclosing environment so they can be imported.*

<name-definition> =
    (**defname** <loader-identifier> <expression>)

<constant-definition> =
    (**defequate** <loader-identifier> <expression>)

<script-definition> =
    (**defpract** <loader-identifier> (<symbol>*) (<loader-identifier>*)
        [([**:machine-dependent**] [**:actor-types** (<actor-type>*)])]
        <communication-handler>*)

<actor-type> = <symbol>
Note: <actor-type> *symbols are not evaluated.*

10.12 Actor Definition

<script-expression> =
    (**script**
     ((<symbol>*) (<symbol>*)
        [([**:machine-dependent**] [**:actor-types** (<actor-type>*)])]
        <communication-handler>*)
      <pract-expression>*)

<communication-hander> =
    (**is-request** <pattern>
        <command>*) ||

```
    (is-request (<pattern>
                  :self <symbol>
                  :sponsor <symbol>
                  :customer <symbol>
                  :reply-keyword <symbol>
                [{ :unserialized | :serialized }])
        <command>*) ||
    (is-reply <pattern>
        <command>*) ||
    (is-reply (<pattern>
                  :self <self-id>
                [{ :unserialized | :serialized }])
        <command>*) ||
    (is-complaint <complaint-pattern>
        <command>*) ||
    (is-complaint (<complaint-pattern>
                  :self <self-id>
                [{ :unserialized | :serialized }])
        <command>*)

<pattern> = (<keyword> <communication-bindings>*)
<complaint-pattern> = (<keyword> <symbol> <communication-bindings>*)

<communication-binding> =
    <symbol> <communication-binding> ||
    &key <communication-binding> ||
    &rest <communication-binding> ||
    &optional <communication-binding> ||
    &allow-other-keys <communication-binding> || ε


<create-expression> =
    (create <pract-expression>      ; new actor's script
        <pract-expression>*)        ; new actor's acquaintances
```

10.13 Pract Commands

```
<command> =
    <send-command> ||
    <let-command> ||
    <if-command> ||
    <replace-command> ||
    <update-command>

<send-command> =
    <request-command> ||
    <reply-command> ||
    <complain-command>

<request-command> =
    (request <pract-expression>             ; target actor
        (<keyword> <pract-expression>*)     ; message name and args
        <pract-expression>                  ; sponsor actor
```

```
        <pract-expression>              ; customer actor
        <pract-expression>)             ; reply keyword
    ||
    (request* <pract-expression>            ; target actor
        (<keyword> <args-nospread>)         ; message name and args
        <pract-expression>              ; sponsor actor
        <pract-expression>              ; customer actor
        <pract-expression>)             ; reply keyword

<reply-command> =
    (reply-to <pract-expression>            ; customer actor
        (<keyword> <pract-expression>*))    ; keyword and values
    ||
    (reply-to* <pract-expression>           ; customer actor
        (<keyword> <args-nospread>))        ; keyword and values

<complain-command> =
    (complain-to <pract-expression>       ; customer actor
        (<keyword> <pract-expression>   ;err & reply keyword
          <pract-expression>*))         ; other values
    ||
    (complain-to* <pract-expression>      ; customer actor
        (<keyword> <pract-expression>   ; err & reply keyword
          <args-nospread>))             ; other values

<let-command> =
    (let ( {(<symbol> <pract-expression>)}* )
        <command>*)

<replace-command> =
    (replace <pract-expression>         ; actor to replace
        <pract-expression>              ; with new behavior script
        <pract-expression>*)            ; and new acquaintance values

<update-command> = (update <pract-expression>    ; actor to update
                {(<acquaintance-number> <pract-expression>)}*)
                                ; new acquaintance values

<acquaintance-number> = [A non-negative integer]

<if-command> =
    (if <pract-expression>
      (then <command>*)
      (else <command>*))
```

10.14 Pract Expressions

```
<pract-expression> =
    <create-expression> || <script-expression> (see behavior defn.)
    <if-expression> || <let-expression> ||
    <bound-identifier> || <condition> || <constant> ||

<if-expression> =
```

(**if** <pract-expression>
(**then** <command>* <pract-expression>)
(**else** <command>* <pract-expression>))

<let-expression> =
(**let** (*{*(<symbol> <pract-expression>)*}**)
<command>* <pract-expression>)

<condition> =
<bound-identifier> ||
(**not** <condition>) ||
(**or** <condition>*) ||
(**and** <condition>*)
[Note: <bound-identifier> *is interpreted as either* NIL *or* (not NIL). *The symbol should be bound either as acq-name, arg identifier, or let binding.]*

<args-nospread> = *[A symbol to which is bound the &rest arguments of a handler.*

<keyword> = <symbol>
[Note: a <keyword> *should be a symbol in the keyword package, e.g.* `:high`, `:low`, *or an identifier bound to a keyword symbol, such as the reply-keyword.]*

<loader-identifer> = [identifer referencing load-time symbol table]

<bound-identifier> = <symbol>

<symbol> = *[a lisp symbol not in the keyword package]*

<constant> = <number> || <quoted-symbol> || <quoted list> ...

Index

Bibliography

[Abelson and Sussman 85]
H. Abelson, and G. J. Sussman, with J. Sussman.
Structure and Interpretation of Computer Programs.
MIT Press, Cambridge, Mass., 1985.

[Agha 86] G. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
MIT Press, Cambridge, Mass., 1986.

[Arvind & Culler 86]
Arvind and David E. Culler.
Managing Resources in a Parallel Machine.
In *Fifth Generation Computer Architectures 1986*, pages 103-121. Elsevier Science Publishers B.V., 1986.

[Clinger 81] W. D. Clinger.
Foundations of Actor Semantics.
AI-TR- 633, MIT Artificial Intelligence Laboratory, May, 1981.

[Dybvig et al. 84] R. Kent Dybvig, Daniel P. Friedman, Christopher T. Haynes.
Expansion Passing Style: An Additional Argument for Macros.
In *1984 ACM Symposium on Lisp and Functional Programming*. 1984.

[Goldberg 83] A. Goldberg and D. Robson.
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley Publishing Company, 1983.

[Halstead 79] Robert H. Halstead.
Reference Tree Networks: Virtual Machine and Implementation.
MIT/LCS/TR- 222, MIT Laboratory for Computer Science, July, 1979.

[Halstead 85] Robert H. Halstead.
Multilisp: A Language for Concurrent Symbolic Computing.
ACM Transactions on Programming Languages and Systems 7(4):501-538, October, 1985.

[Hewitt 77] C. E. Hewitt.
Viewing Control Structures as Patterns of Passing Messages.
Journal of Artificial Intelligence 8-3:323-364, June, 1977.

[Hoare 78] C. A. R. Hoare.
Communicating Sequential Processes.
CACM 21(8):666-677, August, 1978.

[Kahn et al 86] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow.
Objects in Concurrent Logic Programming Languages.
In *Object Oriented Systems, Languages, and Applications 1986 Conference Proceedings*, pages 242-257. ACM SIGPLAN, Association for Computing Machinery, New York, NY, November, 1986.

[Kornfeld 82] W. A. Kornfeld.
Combinatorially Implosive Algorithms.
CACM 25(10), October, 1982.

[Moon 84] D. A. Moon.
Garbage Collection in a Large Lisp System.
In *1984 ACM Symposium on Lisp and Functional Programming*, pages 235-246. 1984.

[Moon 85] D. A. Moon.
Architecture of the Symbolics 3600.
In *The 12th Annual Symposium on Computer Architecture*. IEEE, June, 1985.

[Nikhil et al 86] R. S. Nikhil, K. Pingali, and Arvind.
Id Nouveau.
Computation Structures Group Memo 265, MIT Laboratory for Computer Science. July, 1986

[Pountain 85] Dick Pountain.
A Tutorial Introduction to Occam Programming.
INMOS Limited, Bristol, UK, 1985.

[Rees et al. 86] J. Rees, and W. Clinger, *Eds.*.
The Revised^3 Report on the Algorithmic Language Scheme.
AI Memo 848a, MIT, Cambridge, Ma., September, 1986.

[Shapiro 83] E. Y. Shapiro.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR-003, ICOT, Tokyo, Japan, February, 1983.

[Steele 76] Guy L. Steele.
LAMBDA: The Ultimate Declarative.
AI Memo 379, MIT, Cambridge, Ma., November, 1976.

[Steele 78] Guy L. Steele.
RABBIT: A Compiler for Scheme.
Technical Report AI-TR-474, MIT, Cambridge, Ma., May, 1978.

[Ueda 86] Kazunori Ueda.
Guarded Horn Clauses.
PhD thesis, University of Tokyo, March, 1986.

[US DoD 83] *Reference Manual for the Ada Programming Language*
United States Department of Defense, Washington D.C., 1983.

[Weinreb and Moon 81]
D. Weinreb and D. Moon.
LISP Machine Manual
MIT, 1981.