Functional Real-Time Programming : The Language Ruth And Its Semantics

by

Dave Harrison

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the University of Stirling

September 1988

Abstract

Real-time systems are amongst the most safety critical systems involving computer software and the incorrect functioning of this software can cause great damage, up to and including the loss of life. If seems sensible therefore to write real-time software in a way that gives us the best chance of correctly implementing specifications. Because of the high level of functional programming languages, their semantic simplicity and their amenability to formal reasoning and correctness preserving transformation it thus seems natural to use a functional language for this task.

This thesis explores the problems of applying functional programming languages to real-time by defining the real-time functional programming language **Ruth**.

The first part of the thesis concerns the identification of the particular problems associated with programming real-time systems. These can broadly be stated as a requirement that a real-time language must be able to express facts about time, a feature we have called time expressibility.

The next stage is to provide time expressibility within a purely functional framework. This is accomplished by the use of timestamps on inputs and outputs and by providing a real-time clock as an input to **Ruth** programs.

The final major part of the work is the construction of a formal definition of the semantics of **Ruth** to serve as a basis for formal reasoning and transformation. The framework within which the formal semantics of a real-time language are defined requires time expressibility in the same way as the real-time language itself. This is accomplished within the framework of domain theory by the use of specialised domains for timestamped objects, called herring-bone domains. These domains could be used as the basis for the definition of the semantics of any real-time language.

To my parents :

Jim Harrison (1914 - 1975),

Teresa and Bill Carr

"*...What I want to establish is how accurate Ruth's time sense is. Some dragons don't have any at all*".
"*Ruth always knows when he is*", Jaxom replied with quick pride. "*I'd say he had the best time memory on Pern*"

Anne McCaffrey, "The White Dragon"
Sidgewick & Jackson 1979

Acknowledgements

My grateful thanks to :-

My family for their unfailing support, despite my stubborn refusal to leave university and get a proper job.

My colleagues in The University of Stirling Department of Computing Science for providing such a friendly and stimulating environment in which to work.

Peter "*The Man in Black*" Henderson for giving me the chance to start a Ph.D., and "*Uncle*" Ken Turner for giving me the chance to finish it.

Mustafa Marashi for getting me to work on time, despite an almost total lack of cooperation on my part.

My fellow members of *The Office Cricket Club*, Andrew "*Rumpelstiltskin*" Reeves, Andrew "*Yosser*" Sinclair and Paul "*The Babbler*" Williamson for keeping me relatively sane while I was writing up. This incredible feat was accomplished by allowing me to top the batting averages for a considerable period, and by plying me with large quantities of the products of the Belhaven and Maclay breweries. Thanks also to *The Babbler* for proof reading large parts of this thesis.

Sam "*The Urban Terrorist*" Nelson and the staff and customers of *The Settle Inn* for post-thesis depression counselling.

Simon "*The Caretaker*" Finn for his valiant, and almost successful, efforts to instruct me in the rudiments of domain theory; for proof reading most of this thesis; and for giving helpful comments on the construction of herring-bone domains. Phil Wadler and John Launchbury of Glasgow University also provided valuable suggestions on herring-bone domain construction.

Finally, and most importantly, my supervisor, Simon "*The Boss*" Jones, without who's ideas, enthusiasm and the fairly frequent interactions between his boot and my backside this thesis would never have been written.

Dave "*Baldrick*" Harrison

Contents

Chapter 1 : Defining The Problem : Real-Time Systems And The Requirements For A Real-Time Programming Language

Chapter 2 : A Survey Of Different Approaches To Real-Time Language Design

Chapter 3 : The Language **Ruth**

Chapter 4 : Semantic Domains For Real-Time Programming

Chapter 5 : The Semantic Definition Of **Ruth**

Chapter 6 : Using **Ruth** : A Real-Time Computer Game

Chapter 7 : Review, Assessment And Future Work

Appendix 1 : Formal Presentation Of The Semantic Domains

Appendix 2 : Formal Presentation Of The Semantics Of **Ruth**

Appendix 3 : The Syntax Of **Ruth**

Chapter 1 : Defining The Problem : Real-Time Systems And The Requirements For A Real-Time Programming Language.

1.1 Introduction

Real-time systems are amongst the most safety critical systems involving computer software. The incorrect functioning of real-time software can cause great damage, up to and including the loss of life. It seems sensible therefore, to write real-time software in a way that gives us the best chance of correctly implementing specifications. Because of the high level of functional programming languages, their semantic simplicity and their amenability to formal reasoning and transformation it thus seems natural to use a functional language for this task. However the functional style seems to have made little headway amongst real-time programmers and the question must be asked whether this is due to a basic unsuitability of the functional style of programming for writing real-time software.

The first functional programming language was LISP [McCarthy 60], yet despite functional languages' relatively long history it is only in recent times that they have been regarded as a practical approach to solving real programming problems. Two major reasons have been advanced for this : first, a belief that such languages were, by their very nature, unsuited to solving many classes of programming problems; second, the difficulty of producing efficient implementations of functional languages on traditional von Neumann machines. Because of the nature of the computing industry the second of these has always been considered the most important : ease of writing, debugging and maintaining software has always come a poor second to the speed at which software executes on the hardware.

This ordering of priorities is beginning, albeit slowly, to change because of the rapidly rising cost of software production and the equally rapid fall in the cost of hardware. Even so, efficiency considerations will always be important, particularly in the area which concerns this work : that of real-time systems. However, functional languages need not be less efficient than traditional imperative ones. There is a growing interest in forms of computing engine different from the traditional sequential computer because of an effect which has been called the *von Neumann bottleneck* [Backus 78]. This phrase refers to the fact that the speed of a processor is limited by the speed with which it can communicate (both data and instructions) with its memory.

If the speed with which one processor may execute a particular task is limited, then the obvious way to improve performance is to use more than one processor to execute the task co-operatively (so called *parallel processing*). To fully exploit the potential of parallel processing systems all of the available processors must be kept busy all of the time, or in other words the task must be partitioned into as many sub-tasks as there are processors. This can be difficult with traditional, imperative, languages, such as Pascal and Fortran, which are based on the von Neumann model of executing operations sequentially, each operation making some change to the *state* of the machine. The result of the computation is the final state of the machine. Functional languages, on the other hand exhibit the property of referential transparency. The following definition of referential transparency is given in [Stoy 77].

> "The only thing that matters about an expression is its value, and any expression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same whenever it occurs."

Functional languages do not have the notion of a state which can be changed. A functional program is simply an expression defining the value of the output of the program as a function of its inputs. Because of referential transparency the order in which different parts of a functional program are evaluated cannot change the value of its result. In particular, different sub-expressions of the program may be evaluated at the same time. Thus the potential for parallel evaluation is much easier to realise with a functional language than with an imperative one.

It seems reasonable to hope that by using parallel computer architectures specifically designed for evaluating functional languages (e.g. ALICE [Darlington & Reeve 81], GRIP [Peyton Jones 85]) it will soon be possible for a functional program to exceed the performance of an imperative program running on a von Neumann machine.

Assuming this to be the case it would seem that the only barrier to the use of functional languages for real-time systems work is that of suitability. In other words, how easy is it to express solutions to real-time problems in a functional language : do the difficulties outweigh the advantages? This work attempts to answer this question by presenting the functional real-time programming language **Ruth**, first introduced in [Harrison 87], giving a full denotational semantics for it, and demonstrating its utility.

The first task is to determine what a real-time system is, and in particular what makes real-time systems different from other systems. This issue is addressed in the next section of this chapter which introduces a classification of real-time systems so as to specify exactly what application area **Ruth** is directed towards. Following on from this Section 1.3 derives some basic requirements for a real-time programming language. The final section of this chapter summarises these requirements and presents and overview of the remainder of the work.

1.2 What Is A Real-Time System?

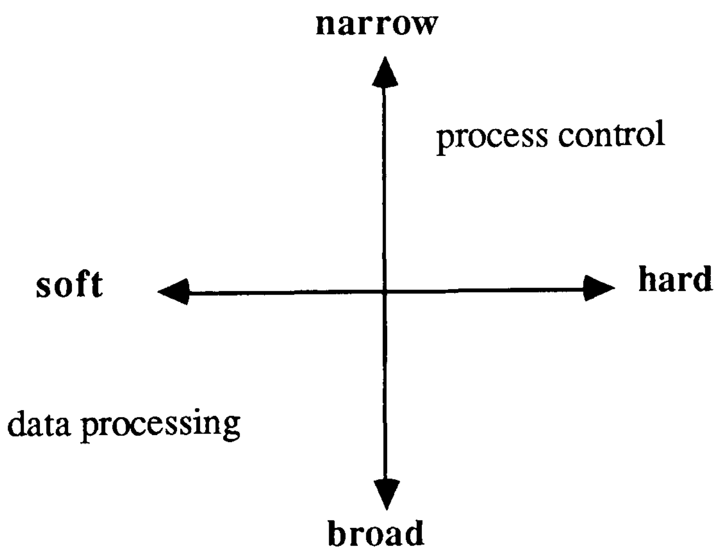
A real-time system can be classified as one in which *when* events occur is as important as *what* events occur. By an event we mean an interaction between the system and its environment. Typical events could be : the input of a value from a keyboard; the output of a control signal to an actuator; or the ticks of a real-time clock. The correctness of a real-time system depends not only upon the values of its inputs and outputs and their relative ordering but also upon their absolute position in time. The situation is very well summarised in [Young 82] :

> "In its broadest sense, the term real-time can be used to describe any information processing activity or system which has to respond to externally generated input stimuli within a finite and specifiable delay."

A real-time system can only be considered to be behaving correctly if it interacts with its environment within specified periods of time, or in other words within its **time windows** (cf. [Faustini & Lewis 86]). The use of the phrase time windows emphasises that systems which interact with their environment too early are just as much in error as those which interact too late. Usually the lower limit of a time window is defined by the arrival of some input from the environment and we are interested only in the upper limit of the window : the time taken by the system to react to the input event (the **deadline**). However the lower limit of a time window is not always fixed in this way and it would be unwise to forget this. For example a rocket engine may be required to cut out 30 seconds after take off. If the engine cuts out too soon the rocket may fail to reach escape velocity.

Real-time systems vary from **broad** systems in which the time windows are large compared to the speed of a software implementation (e.g. electronic timetabling systems) to **narrow** systems in which the time windows are small (e.g. signal processing). The width of a system's time windows is not the whole story however. A further criterion is the likely consequence of a system's failure to respond to events within its time windows. A disk head controller may have only micro-seconds to position the heads on the correct track to access a particular block. If the controller fails to meet this deadline, so that the block has already passed under the heads before they are in position, then the consequences are not serious : the controller need only wait for the disk to complete another revolution before it can access the block. On the other hand it may take a nuclear power station many minutes to become unstable; any control signals during that time could restabilise the situation but if none are forthcoming the reactor will become critical, with disastrous results. Clearly, the disk head controller is the narrower of the two problems but it is much more important that we meet the time windows in the nuclear power station controller. This is the traditional classification of real-time systems into **hard** systems which must always meet their time windows to avoid disaster (such as the nuclear power station controller) and **soft** systems which can tolerate a certain amount of failure (such as the disk head controller). (e.g. [Shin 87])

As Le Lann argues [Le Lann 83], the major difference between what are commonly called real-time systems (the hard end of the range) and conventional data processing systems (the soft end) is the degree to which they can tolerate failures to meet time windows. Whereas we prefer a conventional, soft system to produce some result even if it takes longer than expected, a hard real-time system must produce its outputs within a particular window or not at all. Conversely if a conventional system produces a result earlier than expected this is a benefit; in a real-time system it may well be a disaster as the external environment may not be ready to receive it.



1.1 : **A classification of real-time systems**

Thus we have the classification shown in Fig. 1.1. At the top right are the hard, narrow problems, such as physical process control. At the bottom left are the soft, broad problems such as conventional data processing. Of course this classification should not be taken as totally definitive but with some reservations we can take Fig. 1.1 to be a reasonable view of the situation.

In the main, problems towards the bottom and left of the diagram are assumed to have no real-time aspects. The windows are so wide (i.e. from *now* to *eventually*) and/or the consequences of failure so slight (i.e. try again later) that there is no need to complicate matters by considering *when* as well as *what*. Problems towards the top and right of the diagram are usually handled by custom built electronics. The windows are so narrow (i.e. micro-seconds) that no software system can be expected to cope without generating more window violations than the environment can tolerate. It is problems between these two extremes in which we are mostly interested since these are the systems which are amenable to software monitoring and/or control, provided that the software is written with timing requirements in mind. In the rest of this work this is the type of system we shall call a real-time system.

1.3 Requirements For A Real-Time Programming Language

For a software system to take account of timing requirements essentially requires two things : that the system be able to specify when it wishes events to happen and that it be able to detect when events have (or have not) happened. This obviously requires that a real-time programming language possesses features allowing it to express real-time information. It must be able to express such things as "turn on the fuel injector 20 milliseconds after the engine is switched on" and "if an acknowledgement is not received before a timeout signal abort the communication". Such **time expressibility** is one of the most important requirements for a real-time programming language.

The time windows constraining a real-time system are usually fixed by its need to interact with physical hardware. As pointed out in [Le Lann 83] interactions between software and hardware in real-time systems (called events above) are usually not transparently recoverable. Incorrect events cannot be simply "rolled-back" and forgotten about as they will have produced (potentially disastrous) reactions from the physical hardware (such as moving a control surface on an aircraft). In non real-time systems the user's response to incorrect events is simply to try again but in a real-time system this is usually impossible because of the changes to the environment caused by such events.

The requirement that real-time systems interact correctly with their environment is no different from the requirement for conventional systems, although the failure of a real-time system to meet this requirement is usually much more dangerous. Consequently, language features aiding the writing of correct non real-time software should be equally effective in the real-time domain. [Young 82] gives a set of desirable features for a real-time language, such as strong typing, data abstraction and modular structure which are equally applicable to any programming language. He also makes the point that languages should be as simple as possible since simple languages are easier to learn and programs written in them are easier to understand and reason about.

The major benefit of the use of language features like strong typing, data abstraction and modular structure is that they support structured programming techniques which tend to make programs easier to develop, debug and maintain. They also make it easier for the compiler to detect syntactic and static semantic errors. However, there are limits to the errors that can be detected automatically at compile time. Logical errors may only become apparent at run-time when it may be too late to correct them.

One way in which this situation can be improved is by formal transformation and reasoning. By formal transformation we mean the ability to write simple and clear algorithms which are inefficient to execute on a computer, and then automatically transform them into semantically equivalent, complex, obscure, but efficient, versions for execution (e.g. [Darlington 82], [Burstall & Darlington 77]). Another great aid to detecting logical errors is the ability to formally reason and to prove facts about programs. Formal transformation and proof must be based on a formally defined semantics, which consequently is also one of our requirements for a real-time programming language.

At present program proving is a lengthy and complex task, even for small programs, and even the most rigorously proven program is still vulnerable to hardware failure. It is usually impossible to completely prove a piece of software correct and equally impossible to completely test it, but, as pointed out above, incorrect software behaviour could be disastrous. Real-time programmers are thus forced to program in a very *defensive* manner, particularly in respect of meeting time windows, since it is often better to produce the wrong answer at the right time, or no answer at all, than to produce the right answer at the wrong time. If there is the slightest doubt as to whether a program will meet its deadlines then it is rewritten, and in most real-time systems the time-critical part of the software can be expected to complete its tasks in as little as half the time available to it. Even so, a real-time system is usually written in such a way that an imminent failure to meet a time deadline will be detected and recovery action taken. This requires that the system be able to monitor its own progress in time towards these deadlines : the system must have knowledge of the passage of real time.

The essential requirements for a real-time language are thus :

(i) The ability to define that a particular event should take place at a particular time.

(ii) The ability to detect when an event occurs, and, by extension, the ability to detect whether or not a particular event occurs at a particular time.

(iii) The ability to detect and recover from errors, particularly timing errors.

(iv) A formal semantics allowing transformation and proof techniques to be applied to programs.

The first two of these requirements are what we have called time expressibility and are specific to real-time languages. The third requirement essentially means that the language must support what we have referred to as defensive programming. The major use of defensive programming in real-time systems is to ensure that deadlines are met. Consequently the detection and correction of timing errors is the ability we shall be most interested in. Once again, this ability is specific to real-time languages and is another aspect of time expressibility. Thus the first three requirements may be coalesced into one : that a real-time language have good time expressibility.

The final requirement, for a formal semantics, is applicable to both real-time and non real-time languages. However, because of the safety critical nature of most real-time systems, its importance in the real-time domain cannot be overstressed. Although completely proving programs correct from such a semantics is probably infeasible at present, the safety critical parts of programs could be proven, thus increasing confidence in the reliability of the program. Further, the use of transformation techniques cannot be justified without such a semantics.

1.4 Conclusion

Real-time systems are usually safety critical systems in that the actions they take are generally not recoverable : once something has been done it cannot be undone. Since real-time software is often used to control aircraft, power stations and the like, the correctness of real-time software can be a matter of life and death. The use of functional languages is an aid to the production of correct software because of their relative simplicity and amenability to formal proof and transformation. However there seems to be a belief that functional languages are not suitable for writing real-time software. This work attempts to disprove this by defining, and demonstrating the utility of, **Ruth**, a functional language for writing real-time software.

A real-time problem can be classified as one in which *when* events occur is as important as *what* events occur. The importance of when events occur in a real-time system has two major implications : the system must specify when it wishes events to occur and it must be able to detect when events have (or have not) occurred. Thus, the design of **Ruth** must incorporate features allowing it to handle real-time information : **Ruth** must have what we have called **time expressibility**.

The advantages of formal reasoning and transformation in aiding the production of correct software have been noted above. Without a semantic definition of a programming language formal reasoning and transformation are impossible. Consequently a complete formal definition of **Ruth** will be a major part of this work.

Software engineering issues such as strong typing, modular structure and information hiding are also important in the design of a real-time language. However they are equally important in the design of non real-time languages and consequently have been addressed in the design of several functional languages already (e.g. HOPE [Burstall et. al. 80], [Sannella 81], Miranda [Turner 85] and Haskell [Hudak et. al. 89]). The problems of incorporating software engineering and structured programming into functional languages have been, and will continue to be, extensively researched. In contrast, time expressibility and the semantics of real-time languages have been largely ignored. Consequently it is these issues which will be the major concern of this work.

The next chapter of this thesis contains a survey of the way in which the problems of time expressibility and formal semantics for real-time have been tackled in three different programming paradigms : the imperative, dataflow and functional paradigms. From this basis the concepts underlying the constructs of the language **Ruth** will be distilled; **Ruth** itself is described in Chapter 3. Chapters 4 and 5 introduce the semantic definition of **Ruth** which is based on the denotational framework (e.g. [Stoy 77], [Schmidt 86]). Chapter 6 demonstrates the utility of **Ruth** for writing real-time systems by the construction of a reasonably substantial example : an interactive computer game. Finally, in Chapter 7, we look at what has been achieved and attempt to answer the initial question : is the functional style of programming suitable for solving real-time problems or not?

Chapter 2 : A Survey Of Different Approaches To Real-Time Language Design

2.1 Introduction

The purpose of this chapter is to compare how three different language paradigms, imperative, functional and dataflow, solve the special problems associated with the programming of real-time systems which were discussed in the opening chapter of this thesis. The comparison will concentrate on the time expressibility offered by languages within the three paradigms and on the existence of formal semantic models, since, as commented at the end of Chapter 1, these are areas which have received little attention in the past.

The next section of this chapter concerns imperative languages, in particular Ada, thus allowing the evaluation of current industrial practice with respect to time expressibility and formal semantics. Section 2.3 concerns an area of current research : real-time languages based on the dataflow model. Section 2.4 assesses real-time and related work in functional programming to determine what basis exists from which to construct **Ruth**. The concluding section of this chapter brings together these different strands to determine what problems remain with the application of the functional style to real-time programming.

2.2 The Imperative Approach

Most computer programming languages are imperative. A program written in such a language is a sequence of commands telling the computer how to change its state so as to arrive at the final solution. Imperative languages directly model the workings of the von Neumann computer, indeed their basic, sequential style of execution was derived from it.

Because of its adoption as the required implementation language for real-time systems by the US Department of Defense and the UK Ministry of Defence probably the most important imperative real-time language is the language Ada [USDOD 83]. Since the defence industry is the world's major producer and consumer of real-time software it seems likely that Ada will be the major language used in real-time work for the foreseeable future.

Ada's time expressibility is based upon the predefined package CALENDAR which provides the type TIME and operations to manipulate that type, including a CLOCK primitive which returns a representation of the current time in seconds. Ada also supplies the **delay** primitive to suspend execution of a task for (at least) a given period. The example below is taken from [USDOD 83] and causes an event to be repeated at intervals of not less than INTERVAL seconds.

```
declare                                                  (2.1)
  use CALENDAR;
  -- INTERVAL is a global constant
  NEXT_TIME : TIME := CLOCK + INTERVAL;
begin
  loop
    delay NEXT_TIME - CLOCK;
    -- some event
    NEXT_TIME := NEXT_TIME + INTERVAL;
  end loop ;
end;
```

The interval between occurrences of the event is only approximately equal to INTERVAL because the **delay** primitive only guarantees a minimum waiting time. In the example above the task could be kept waiting for much longer than INTERVAL if other tasks are sharing the same physical processor. There is no way of specifying in Ada that an event must occur at a particular time. It is straightforward to specify that it must not occur *before* a particular time, but the Ada programmer has no control over how long *after* this time the event occurs.

Events are detected in Ada via the **rendezvous** mechanism. A real-time program written in Ada is composed of one or more **tasks**, or processes, running in parallel. Communication between tasks, and thus between an Ada program and its environment, is via the rendezvous mechanism : a task may call an **entry** in another task and then wait until the called task is willing to accept the call. When the call is accepted the tasks are said to have rendezvoused and, after the code associated with the entry has been executed, the tasks part and continue their independent execution.

A task may accept any one of a number of calls from other tasks by using a **select** construct. The example below will accept the events INC, DEC and CLEAR, each of which causes the value of the variable count to be changed in the obvious way.

```
select                                                    (2.2)
  accept INC;
  count := count + 1;
or
  accept DEC;
  count := count - 1;
or
  accept CLEAR;
  count := 0;
end select;
```

Whenever another task (or the hardware environment) issues a call to either INC, DEC or CLEAR a rendezvous takes place and the count variable is updated. If there is a call to more than one event already present when the **select** statement starts executing then an arbitrary event is accepted. If there are no calls waiting then the **select** waits until a call occurs.

The behaviour of **select** when there are no calls waiting initially is an example of what we shall call **implicit time determinance**. In this situation **select** accepts the *first* call that arrives after it starts executing. The choice of which call to accept appears non-determinate from the text of the program but is not so if the implicit argument to the **select**, time, is taken into account. Non-determinance is a powerful tool for abstracting away from irrelevant details; however it seems somewhat bizarre to use non-determinance to abstract away from timing information in a real-time program, particularly since timing information constitutes half, and probably the more important half, of the information available.

A further problem with **select** is its behaviour if there are calls already waiting when it starts execution. In this case the called task decides to accept an arbitrary call and the basis for this decision is not defined by the language. Once again this abstraction seems somewhat bizarre in a real-time language. If the programmer does not know which call will be accepted then he has no alternative but to program defensively to ensure that the correct priority in accepting calls will be adopted.

Ada allows programs to detect the non occurrence of an event by a particular time via the selective wait construct.

```
select                                                    (2.3)
  accept MESSAGE;
or
  delay 10*MILLI_SECONDS;
  TIMEOUT;
end select;
```

If the MESSAGE event occurs within 10 milliseconds of the start of execution of the **select** then TIMEOUT is avoided. Obviously, by using the CLOCK primitive, absolute times can be referenced in the **delay** as well as relative times as above.

The CLOCK primitive also allows Ada programs to monitor their progress in time and thus to check whether deadlines are going to be met. An alternative approach is offered by the timed entry call which allows a calling task to abort a call if it is not accepted by a certain time.

```
select                                          (2.4)
  MESSAGE;
or
  delay 10*MILLI_SECONDS;
  TIMEOUT;
end select;
```

If the entry call MESSAGE is not accepted within 10 milliseconds of the start of execution of the select then TIMEOUT results. By using either of these methods an Ada program can detect, and thus recover from, timing errors.

Ada is an important, probably the most important, real-time programming language because of its adoption by the US Department of Defense and UK Ministry of Defence as the standard implementation language for real-time applications and any realistic survey of real-time languages must reflect this.

In many ways Ada can be regarded as the epitome of a language supporting structured programming. It is strongly typed with a very rich type scheme and the package construct allows good data abstraction and modular programming. Because Ada was designed to fulfil all the US DoD's software requirements, and not just those in the real-time domain, it is a very large and complex language. The complexity of Ada has made the definition of a formal semantics and the production of transformation rules very difficult (e.g. [Bjorner & Oest 80]). One of the major problems which anyone attempting a semantics for Ada has to address is the use of the implicit time determinate select construct. A more serious criticism of Ada is that its time expressibility is seriously flawed. Detecting that an event has not occurred by a particular time is fairly straightforward as is the detection and recovery from potential and/or actual timing errors. However, there is no way to specify that an event must occur at a particular time. For a real-time language not to have this basic capability is a serious design flaw.

These problems are not specific to Ada but appear, to a greater or lesser extent, in most imperative languages used for real-time programming (e.g. CORAL-66 [MoD 70], and Modula-2 [Wirth 83]). In the next section we shall look at a group of languages which are based not on the von Neumann model but upon the dataflow model of computation.

2.3 The Dataflow Approach

2.3.1 Lucid

The dataflow language Lucid [Ashcroft & Wadge 76,77,80], [Wadge & Ashcroft 85] is both a programming language and a notation for formal reasoning. The basic principle of Lucid, in keeping with the dataflow model, is that identifiers and constants in Lucid do not denote single values but infinite, indexed streams of values or **histories**. For example, the constant 1 in Lucid denotes the history <1,1,1,1, ...>. The identifier x denotes a history containing successive values for x and is often written <x_0,x_1,x_2, ...>. Operators in Lucid work pointwise upon histories, for example

Let x = <0,2,4,6,8,10, ...> (2.5)
and y = <1,3,5,7,9,11, ...>
then x + y = <x_0 + y_0,x_1 + y_1, x_2 + y_2, ...>
= <1,5,9,13,17,21, ...>

For constructing histories from other histories Lucid has the **fby** (followed by) operator which is similar to **Cons** in functional languages. **fby** constructs a new history by adding the first element of its first history argument to the front of its second history argument .

Let x = <0,2,4,6,8,10, ...> (2.6)
and y = <1,3,5,7,9,11, ...>
then x **fby** y = <0,1,3,5,7,9,11, ...>

The first element of a history can be discarded via the **next** operator.

Let x = <0,2,4,6,8,10, ...> (2.7)
then **next**(x) = <2,4,6,8,10, ...>

Lucid is a referentially transparent language. Lucid programs are simply functions from input histories to output histories. Consequently it has a simple formal semantics which allows relatively easy reasoning and transformation.

2.3.2 Real-time Lucid

The basic Lucid paradigm has been extended in many directions. Indeed Lucid is not just one language but a family of languages and, as might be expected, the area of real-time programming has not escaped the attention of the Lucid community. Real-time Lucid [Faustini & Lewis 85, 86], adds time expressibility to basic Lucid by associating with each value history (that is each identifier or constant's history) a **time window** history defining when elements in the value history should be computed. Each element of a value history's window history is a pair `[l,u]`; the n^th^ element of the value history must not be produced before the time specified by the `l` value of the n^th^ element of the window history, but must be produced before (or at) the time specified by the `u` value of the n^th^ element of the window history. Windows are associated with value histories via the `@` operator.

```
Let  x = <0,2,4,6,8, ...>                                          (2.8)
then x @ [x, next(x)]
        = <0,2,4,6,8, ...> @ <[0,2],[2,4],[4,6],[6,8],[8,10]... >
```

The first element of `x` must be produced between `0` and `2` (inclusive), the second between `2` and `4`, and so on. The window associated with an output value denotes the ranges within which each element of the output value's history is to be delivered to the external environment. The window associated with an input value denotes the ranges within which the real-time Lucid program will sample the input line.

Only input and output value histories are given "real" window histories by the programmer. Histories which are purely internal to the program are given the default window history in which all elements are [-∞, +∞], indicating that the values may be produced at any time that is consistent with the input/output windows.

The time window mechanism is a very powerful one since it allows the real-time Lucid programmer to directly express facts about the required temporal behaviour of his real-time system. For example, that the value `x` is to be output (i.e. the event denoted by `x` is to occur) every `10` time units ±`3` time units would be expressed as overleaf.

```
x @ [index*10 - 3, index*10 + 3]                                  (2.9)
where
index = 1 fby index+1 ;
```

The first output of x is in the window [7,13], the next within [17,23] and so on. If it were required that x be output *exactly* every 10 time units then a *point* window could be used.

```
x @ [index*10, index*10]                                          (2.10)
where
index = 1 fby index+1 ;
```

The n^th output of x is in the window [n*10,n*10]. Since the earliest the output can be produced (i.e. the event the output denotes can occur) is n*10 and the latest it can be produced is n*10 then it must be produced (the event must occur) at exactly n*10.

For detecting when an event occurs real-time Lucid supplies the **time** primitive.

Let x be as defined in (2.10) above (2.11)
then **time**(x) = <10, 20, 30, 40, ..>

The problem with **time** is that its result cannot be determined from the text of the program unless (as above) it is only applied to a value associated with a point window. Otherwise the n^th element of the history returned by **time** may be anywhere between the n^th element of the corresponding l and u histories.

Let x be as defined in (2.9) above (2.12)
then **time**(x) = <10±3, 20±3, 30±3, 40±3, ..>

Since the result of **time** may be used as a normal value in a real-time Lucid program this non-determinance makes a formal semantics difficult to construct, though work on such a semantics is in progress. In fact, **time** is another example of implicit time determinance; if we knew *when* results were computed it would be totally determinate.

Detection of the non-occurrence of an event by a particular time depends on the meaning real-time Lucid actually gives to input and output time windows. As mentioned earlier, the window associated with an input value denotes the ranges within which the real-time Lucid program should sample the input line. If no value is present on the input line during a particular range (i.e. the input event has not occurred) then the program is supplied with a time fault value, written **`tf`**.

Time faults are detected via the boolean operator **`istf`**. In the following example let `i` be an input of "`blip`" values denoting that an event has occurred, and let us assume that the first four `i`'s are actually available to be input at times `0`, `3`, `4` and `7`.

Let `w = [index,index+2]` (2.13)
and `index = 1` **`fby`** `index+1`

index	w	**time**(i)	i @ w	**istf**(i @ w)
1	[1,3]	0	**tf**	true
2	[2,4]	3	blip	false
3	[3,5]	4	blip	false
4	[4,6]	7	**tf**	true
...	...	...	...	...

The time fault at `index=1` in `i @ w` is caused by the input `i` arriving too soon; the time fault at `index=4` is caused by it arriving too late.

Also as mentioned earlier the window associated with an output value denotes the ranges within which each element of the output value's history is to be delivered to the external environment. Time faults are inserted into output histories whenever the program fails to produce the desired value within the specified window. This is a simple form of timing error detection and recovery. Were the language to allow windows to be associated with internal values as well as input/output values the situation could be greatly improved by adding a *supervisor* to each output to detect when time faults are about to occur and take corrective action. An example of this is shown overleaf.

```
out @ w1                                              (2.14)
where
out   = If istf(temp)
          Then default_value
          Else temp ;
temp  = "expression" @ w2
w1    = [0, (index * 4) + 1] ;
w2    = [0, (index * 4)] ;
index = 1 fby 1 + index ;
```

If the result, `temp`, is not computed by the time specified in the "timeout" window `w2` then a default value is substituted for output within the (wider) window `w1`. There seems no reason why windows could not be associated with internal values in this way. However, as real-time Lucid stands, the programmer has no option but to assume that the environment will correctly handle any time fault values produced.

The production of time faults on an input history can be determined from the time the inputs occur and the specification of the time window associated with the input. The production of time faults on an output can be determined from the time the outputs are produced and the specification of the time window associated with the output. Thus the production of time faults is another example of implicit time determinance. If the time at which inputs and outputs occur is taken into account then time fault production is determinate; if not it is non-determinate, making reasoning and transformation of programs very difficult.

The time expressibility of real-time Lucid is very good. Time windows are a direct model of the real-time requirements for a program. The programmer can specify when events should occur, either at a point in time or within a range of times. By using the **time** operator the programmer can also detect when events actually did occur. However the result produced by **time** cannot be determined from the text of a program unless it is applied to events associated with point windows. The time fault mechanism allows the detection of the non-occurrence of an event and supplies a rudimentary form of timing error recovery.

Although Lucid has a fully formal semantics and proof system associated with it [Ashcroft & Wadge 76] there is, at present, no fully formal semantics for real-time Lucid. This may well be due to the complexity of defining both the **time** primitive and the time fault mechanism. Unless a semantics is given some explicit notion of real time both these features will be non-determinate, making the specification of a semantics much more complex.

2.3.3 LUSTRE

Real-time Lucid allows the user to specify when, within a range, interactions with the environment must occur. This is done by associating a window history with each value history. An alternative approach would be to have the indexes in the value history actually specify a fixed time for such interaction. In other words, that the n^{th} element in an input (output) history be input (output) at time n.

This is the approach taken by the language LUSTRE ([Bergerand et. al. 85, 86], [Caspi et. al. 87]) which shares the Lucid view of values as histories, but interprets the position of an element in a history as defining the time at which the event it denotes occurs. Thus the time expressibility in LUSTRE is based totally upon the position of elements within histories.

The real-time interpretation of history positions has one major consequence. Consider the following LUSTRE expression in which `in` is an input history and `out` an output history.

```
out = in + 1                                              (2.15)
```

As in all Lucid-like languages the n^{th} element of the `out` history is obtained by adding `1` to the n^{th} element of the `in` history. However, in order to be consistent with the meaning of an element's position the n^{th} element of `in` must be input at time n and the n^{th} element of `out` must be output at time n. In other words input and output must be simultaneous and thus the addition must take zero time. This is the called the **strong synchrony** hypothesis : all operators are assumed to react instantly to their inputs. On the surface the strong synchrony hypothesis seems unrealistic as it requires computers to be infinitely fast. In fact, all that is required is that the computer is fast enough that the external environment does not notice that it is not infinitely fast. For example, a video game may be receiving input from a user every tenth of a second; provided the system reacts to that input within, say, a fiftieth of a second, the user will never notice that the system is not responding to his input "at the same time" as he is providing it. In other words, provided that the problem is sufficiently broad compared to the speed of the software the strong synchrony hypothesis will cause no problems.

Instead of **fby** and **next** LUSTRE provides the operators -> and **pre**. The major reason for this change is that Lucid's **next** is *non-causal*. That is, the result of **next**(input) could not be decided from the values input at time n, the system would have to wait until time n +1 to obtain the n^{th} element of the **next**(input) history. This obviously violates the strong synchrony hypothesis. **pre** and -> are *causal* operators : their result element at time n can always be determined with information available at time n.

```
Let  x = <0,2,4,6,8, ...>                                    (2.16)
and  y = <1,3,5,7,9, ...>
then
pre(x) = <Nil,0,2,4,6,8, ...>
x -> y = <0,3,5,7,9, ...>
```

pre inserts Nil at the start of its history argument, thus acting as a delaying operator. -> replaces the first element of its second argument history with the first element of its first argument history. Note the following equivalences between Lucid and LUSTRE operators.

```
x fby next(y)      ⇔ x -> y                                  (2.17)
x fby y            ⇔ x -> pre(y)
```

In LUSTRE the position of an element in a history denotes the time at which the event it denotes occurs. Thus the event denoted by the n^{th} element in a history occurs at time n, or in other words at the n^{th} "tick" of what LUSTRE calls the basic clock.

LUSTRE also allows values to have their timing dependent on clocks other than the basic clock. A user defined clock is simply a boolean valued history, a tick being the element true and a "non-tick" being denoted by false. The basic clock is thus the value history denoted by true. For example a clock ticking at half the rate of the basic clock is defined below.

```
c = true -> not(pre(c))                                    (2.18)
  = <true, false, true, false, ...>
```

For any value history x and clock c the operation x **when** c masks off all elements of x for which the corresponding element of c is false. Thus, the result of x **when** c is a value whose clock is c rather than the basic clock.

x	c	y = x **when** c	(2.19)
0	true	y_0 = 0	
1	false		
2	true	y_1 = 2	
3	false		
4	true	y_2 = 3	
5	false		
6	true	y_3 = 5	
...	...	...	

Events in the history y occur at ticks 0, 2, 4, and 6 of the basic clock. The value of y at basic clock tick 5 has no more meaning than the value of x at basic clock tick 1.5. y does not exist at basic clock tick 5 any more than x exists between basic clock ticks.

Since clocks are simply boolean valued histories the LUSTRE programmer can treat them in the same way as any other value which gives a very expressive notion of time. In particular a clock must itself have a basic clock, thus allowing hierarchies of clocks based upon each other to be constructed. For example, assuming that the basic clock ticks every millisecond, a clock that ticks every second can be defined as below.

```
count  = 0 -> If pre(count) ≥ 999                          (2.20)
              Then 0
              Else pre(count) + 1 ;
second = If count ≥ 999 Then true Else false ;
```

Since count is clocked on the basic, millisecond, clock second will be true every 1000 milliseconds, or in other words, every second. In the same way a minute clock could be based upon second, an hour upon minute and so on. To define that an event computed by the expression E should occur every second (i.e. that E is based upon the second clock rather than on the basic, millisecond, clock) the LUSTRE programmer writes

```
E when second                                                    (2.21)
```

At all times other than when second is true the value of E is "masked off". Thus E **when** second only has a value at exactly one second intervals.

By constructing and using clocks in this way it is simple for a LUSTRE programmer to define exactly when events should occur. If the clock upon which the event is based is true then the event occurs and if it is false then the event does not occur. This also makes the detection of the non-occurrence of an event simple. If an event does not occur at or before time n then the clock on which that event is calculated will be false up to time n. For example, assume that the value history denoting the event we are interested in is based on the clock c. The following code fragment detects whether or not the event occurs at n millisecond intervals.

```
count  = 0 -> If c Then 0 Else pre(count) + 1 ;                  (2.22)
status = If count ≥ n Then "timeout" Else "ok" ;
```

Since c is based upon the basic, millisecond, clock, n successive false values mean that n milliseconds have passed without the event occurring. Thus status becomes "timeout". This example serves to illustrate one minor inconvenience caused LUSTRE's implicit notion of time. There is no way of talking about time directly, but only as the number of true ticks on a particular clock. Whether this inconvenience would cause problems when writing large systems is an area for further research.

A much more serious problem is caused by the strong synchrony hypothesis itself. This hypothesis is a very powerful one for the specification of real-time systems : it allows the user to ignore the time delays inherent in computation and concentrate on the requirements of the external environment. However there are problems with actually implementing real-time systems in a language such as LUSTRE because of the underlying assumption of infinite execution speed.

Provided the problem is sufficiently broad, or in other words that the implementation of the language is sufficiently fast, strong synchrony poses no difficulties. Efficient implementations of LUSTRE are possible by compiling the control structure of a program into a finite state automaton [Caspi et. al. 87], although there are problem with distributing such implementations across parallel architectures. But even the most efficient implementation will eventually prove too slow to cope with the deadlines required by a particular environment. Because of strong synchrony LUSTRE has no way of coping with such situations; in fact there is no concept in the language or its underlying proof scheme of a timing error. Since all operations take zero time there is no way in which a value can take too long to compute.

LUSTRE thus fails our third requirement : it cannot detect and recover from timing errors. Nonetheless, for a large class of real-time system in which software is so much faster that its environment that timing errors will not occur languages based on strong synchrony provide an elegant and expressive vehicle for implementation.

A fully formal semantics exists for LUSTRE, including a clock semantics allowing reasoning about temporal behaviour. (See [Caspi et. al. 87] for an introduction). Since LUSTRE, like Lucid, is referentially transparent and totally determinate this semantics is fairly simple and formal reasoning and transformation are relatively straightforward.

Other data flow based languages employing the strong synchrony hypothesis include SIGNAL ([Le Guernic et. al. 85, 86], [Le Guernic & Benveniste 87]) which has a much richer clock semantics associated with it than LUSTRE, and the imperative language ESTEREL ([Berry & Cosserat 84], [Berry et. al. 86, 87a, 87b], historically the first synchronous language. The synchronous languages offer a simple, and very expressive, model of time. However this model depends on an assumption, the strong synchrony hypothesis, which cannot be sustained in the real world and which makes the construction of defensive programs impossible.

In the next section we shall go on to look at our particular area of interest, functional languages.

2.4 The Functional Approach

The major area of interest for this work is that of functional languages. In Chapter 1 we commented that there were two main reasons why functional languages had not been used in the real-time field : their perceived inefficiency and the assumption of their basic unsuitability. The efficiency of functional languages may soon equal, and may eventually surpass, that of imperative languages, not least because referential transparency increases the possibilities for parallel execution of functional programs. The problem, then, seems to be that functional languages are not considered suitable for real-time programming.

Functional languages offer many advantages over imperative ones, particularly in safety critical areas such as real-time, because they are referentially transparent and totally determinate. This greatly aids reasoning about programs and the production of transformation rules (e.g. [Backus 78], [Burstall & Darlington 77], [Darlington 82]). The simpler the semantics of a language the easier it is to understand programs written in it and the more confidence a user will have that they are correct. Finally, functional languages, having a higher level of abstraction than imperative ones, are in general more concise. In general, shorter programs are easier to read and understand

2.4.1 Streams and non-determinance

There has been little research on real-time programming with functional languages but quite a substantial body of research has been carried out into the related area of systems programming with functional languages (e.g. [Abramsky & Sykes 85], [Henderson 82], [Holmstrom 83], [Jones 84a, 84b], [Jones & Sinclair 89], [Stoye 86], [Turner 87]). The major difference between systems programming and real-time programming is that the former only requires knowledge of the relative ordering of events whilst the latter requires knowledge of their absolute position in time (what we have referred to as time expressibility). To support systems programming two basic ideas have been used : stream processing and implicit time determinance. We shall refer to these languages as the implicit time determinate languages.

Streams are a common way in which functional languages model input and output to and from a program's environment (e.g. [Ida & Tanaka 83,84]). For example, the following function takes an input stream of integers as its argument and increments each element by one to produce the output stream.

```
output = f(input)                                                    (2.23)
where
f = lambda (in).Stream_Cons(Head(in)+1, f(Tail(in))) ;
endwhere
```

Note that **Stream_Cons** is a head strict operator : it fully evaluates **Head**(in)+1 before consing it to the front of the rest of the output but does not attempt to evaluate f(**Tail**(in)) before performing the cons. Streams are very similar to Lucid histories save that the programmer must explicitly construct streams and reference their elements in his program. In Lucid this is implicit so that (2.23) would be written

```
output = input + 1                                                   (2.24)
```

Implicit time determinance is used to decide between events (such as keyboard inputs) on the basis of which happens first. What is at issue is the temporal ordering of these events and not, as would be the case in real-time, exactly when the events actually occur. The constructs used vary but, in general, can all be reduced to the **amb** operator first used in [McCarthy 63]. McCarthy states that **amb** is a purely non-determinate binary operator, returning either of its arguments at random. However, when used in the implicit time determinate languages **amb** is always assumed to return the first of its arguments that completes evaluation. Only if both arguments are already completely evaluated when **amb** is applied is the choice potentially non-determinate and in fact, even in this case, the choice is usually implemented as a fully determinate one : an implementation of **amb** usually polls its arguments in order to check whether their evaluation has completed; the first arguments polled will be returned and polling is always carried out in the same order.

The following example shows implicit time determinate **amb** being used to define the function `merge` which interleaves two input streams, `i1` and `i2`, in the order in which the elements of those streams are computed (so called *timewise* merge).

```
merge                                                         (2.25)
  = lambda (i1,i2).
      alt1 amb alt2
      where
      alt1  = If i1 = Nil
              Then i2
              Else Stream_Cons(Head(i1),merge(Tail(i1),i2) ;
      alt2  = If i2 = Nil
              Then i1
              Else Stream_Cons(Head(i2),merge(i1,Tail(i2)) ;
      endwhere
```

Here `Nil` is the usual empty stream marker. `alt1` and `alt2` cannot determine whether their respective input streams are `Nil` until either a stream input or the `Nil` marker is input. If a `Nil` is input then the alternate stream is returned; if a stream input is found then it can be returned and the `merge` continued.

In fact timewise `merge` and implicit time determinate **amb** are basically equivalent operators since **amb** can be trivially implemented using `merge`.

```
amb = lambda (v1,v2).                                         (2.26)
        Head(merge(Stream_Cons(v1,Nil), Stream_Cons(v2,Nil)))
```

Generally speaking all the implicit time determinate languages use either **amb**- or `merge`-like operators.

The basic problem with the use of **amb** in functional programs is its non-determinate semantics. Non-determinance is difficult to deal with theoretically since expressions denote sets of potential results and a powerdomain treatment is required (e.g. [Apt & Plotkin 81], [Apt & Olderog 83], [Broy 82]); this makes proof and transformation very difficult. Further non-determinance also loses referentially transparency : evaluations of `1 amb 2` do not always produce the same result as evaluations of `1 amb 2`; this makes programs difficult to understand and familiar transformation rules ([e.g. Darlington 82]) are not valid.

In cases where non-determinance is required this is something that programmers must learn to live with, but in systems programming, and by extension, real-time programming, *non-determinate* behaviour is not the property that is required. In general programmers do not require **amb** to be non-determinate, they require that it return the first of its arguments to become evaluated. However, because most language semantics abstract away from time, language definers are usually reduced to defining **amb** as a non-determinate operator and making vague, natural language, statements about its real-time properties.

2.4.2 Herring-bone domains and ART

By considering time within a language's semantics it is simple to give a perfectly determinate definition of **amb**. In [Broy 83] the language ART (Applicative language for Real-Time programming) is given a denotational semantics in just this way. Each element of the domain of values computed by programs (including ⊥, the undefined value) is given a timestamp denoting when it was computed (or in the case of ⊥ that it has not been computed by this time). Instead of expressions evaluating to simple values, they evaluate to a pair `<t,v>` where `v` is the data value and `t` is the time at which it is computed. For example `<10,true>` denotes that the boolean value `true` which was computed at time `10` whereas `<10,⊥>` denotes a value that has not been computed at time `10`.

Fig `2.27` overleaf is a diagram of the domain of booleans timestamped in just this fashion. The shape of this diagram leads us to call such domains *herring-bone* domains. In a sense time can be said to "flow" up a herring-bone domain; as computation proceeds the semantic timestamps of ⊥ values become greater and greater until eventually the value is computed and the semantic timestamp becomes fixed, or, if the value is never computed, the `<∞,⊥>` element (undefined at time infinity) results. As commented in [Broy 83] it is a pleasing feature of herring-bone domains that the non-terminating computation is not modelled by the weakest element, `<0,⊥>`, as is usual in normal domains, but by the "permanently undefined" element, $<\infty,\perp>$. In the non real-time domain any program which has not yet produced results conveys the same amount of information as any other. When dealing with real-time a program which has produced no results at time 10 conveys more information than a program which has produced no results at time 5 : the behaviour of the program between times 5 and 10 is now known. $<0,\perp>$ denotes a computation which has not yet begun, and has thus produced no results; $<\infty,\perp>$ denotes a computation which produces no results even after infinite time. Clearly $<\infty,\perp>$ is the element which represents non-termination in a herring-bone domain.

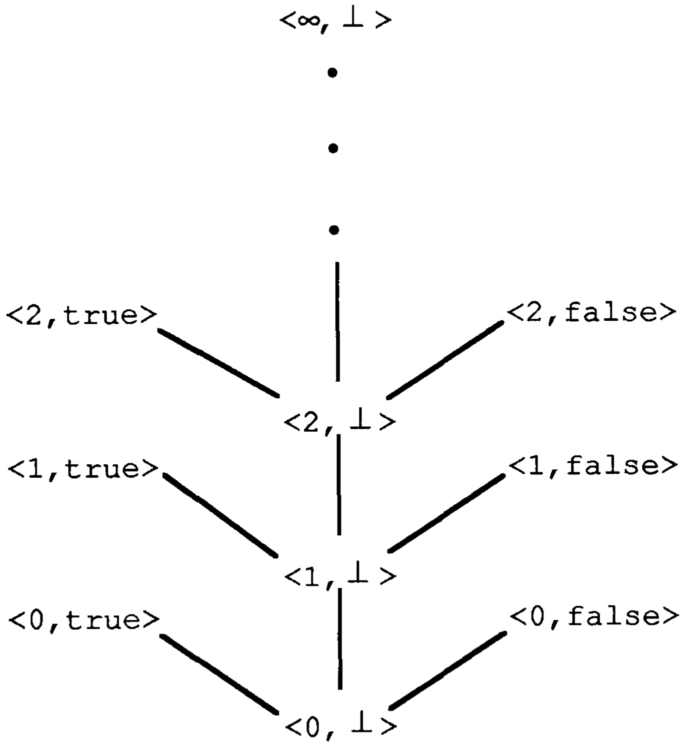


Fig 2.27 : **The herring-bone domain of boolean values**

Using herring-bone domains a determinate definition of **amb** is quite straightforward.

$$<t_1,v_1> \textbf{ amb } <t_2,v_2> \tag{2.28}$$

$$= \begin{cases} <t_1 + 1,\ v_1> & \text{if } v_1 \neq \perp \text{ and } t_1 \leq t_2 \\ <t_2 + 1,\ v_2> & \text{if } v_2 \neq \perp \text{ and } t_1 > t_2 \\ <\min(t_1,t_2) + 1,\ \perp> & \text{otherwise} \end{cases}$$

Note that we do not assume the strong synchrony hypothesis here : to allow for the time taken for the **amb** to be evaluated we increment the timestamp of the result by one. The same approach is taken in the semantics of ART. If only one of **amb**'s arguments is defined ($\neq \perp$) then that argument is selected. If both arguments are defined then **amb** selects the one with the lower timestamp, or the left argument if the timestamps are equal. If neither argument is defined then the result of the **amb** is still undefined at the minimum of t_1 and t_2 (plus one to allow for the **amb**). This is because $min(t_1,t_2)$ is the last time about which we have certain knowledge. If t_1 is less than t_2 then v_1 may become defined before t_2. Likewise, if t_2 is less than t_1 then v_2 may become defined before t_1. Consequently all we can safely say is that the result of the **amb** is definitely undefined at $min(t_1,t_2)+1$.

amb is not an ART primitive, instead the related operator **before** which returns `true` if its first argument is computed before its second, is used. The definition of **before** is given later in this section.

Although **amb** can now be given a determinate semantics we should note that it is not sufficiently discriminating for use in a real-time language and that additional operators will be required. **amb** can determine which of two events happens first, it cannot determine when events actually occur and this is a basic requirement for a functional programming language. The means by which **amb** was given a determinate definition, the herring-bone domains, will, however, prove very useful in this work since they provide a straightforward way for a denotational semantics to express information about time and a formal semantics was also a fundamental requirement for the language. A fuller explanation of herring-bone domains and their properties will be given in Chapter 4 where they will form the basis for the full semantic definition of the language **Ruth**.

The other fundamental requirement was that a real-time language should have good time expressibility. The implicit time determinate languages, with one exception which will be discussed later, have no time expressibility. This is unsurprising since they were not designed with real-time problems in mind. ART was designed as a real-time language and has two primitives which deal with time information. The **delay** primitive, shown overleaf, delays the computation of its first argument until, at least, the time specified by its second argument.

$$\begin{array}{ll} \text{Let} & x = \langle t_1, v_1 \rangle \\ \text{and} & y = \langle t_2, v_2 \rangle \\ \text{then} & \textbf{delay}\ x\ \textbf{for}\ y = \langle \max(t_1, t_2, v_2) + 1,\ v_1 \rangle \end{array} \qquad (2.29)$$

A more mnemonic syntax for this construct might be **delay** x **until** y. Note that the timestamp of the result is incremented by one to allow for the time taken to evaluated the **delay**. The result of the **delay** becomes available at v_2 provided that v_2 is greater than t_1 and t_2, the times at which the two arguments to **delay** are computed. Otherwise it becomes available immediately execution of the **delay** is completed. Thus, so long as v_2 denotes a time later than t_1 and t_2, the ART programmer can use **delay** to define when events are to occur. Unfortunately ART supplies no operations which can detect when a value is computed so that there is no way to check on the relative values of v_2, t_1 and t_2. Consequently, using **delay** to specify when events are to occur is unreliable; an event cannot occur until the values defining it are computed and if this is later than the time the event was due then the event will not occur on time.

As mentioned above, instead of **amb** ART uses the operator **before** which tests which of its two arguments became defined with the lower timestamp (i.e. soonest).

$$\begin{array}{lll} \langle t_1, v_1 \rangle\ \textbf{before}\ \langle t_2, v_2 \rangle & & \\ \quad = \langle t_1 + 1,\ \mathrm{true} \rangle & \text{if } v_1 \neq \bot \text{ and } t_1 \leq t_2 & \\ \quad \langle t_2 + 1,\ \mathrm{false} \rangle & \text{if } v_2 \neq \bot \text{ and } t_1 > t_2 & \\ \quad \langle \min(t_1, t_2) + 1,\ \bot \rangle & \text{otherwise} & \end{array} \qquad (2.30)$$

before, like **amb**, can detect the relative ordering of events and by using a combination of **before** and **delay** timing errors can also be detected.

```
If timeout before result                                    (2.31)
Then timeout
Else result
where
timeout = delay default_value for deadline ;
result  = ... ;
endwhere
```

If `result` is not computed before `timeout` then `default_value` will be returned. `timeout` will not be computed until at least `deadline` but it may well be computed much later than this, for example if evaluation of the above code starts so close to `deadline` that `default_value` cannot be computed in the remaining time. Once again, the use of **`delay`** is problematic because a programmer cannot be sure *when* it will delay its result until.

Since the semantics of ART define exactly when results are produced from expressions there is a solution to this problem : the result of a **`delay`** can be determined by mathematically analysing the program containing it. Thus, although a program cannot detect at run time whether or not time deadlines will be met, since ART has no primitive for detecting when values are actually computed, deadline failures can be proven not to occur in a program from the semantics of the language.

However, ART's assumption that it is possible to give a fixed duration for each machine operation may be no more realistic than the strong synchrony hypothesis due to the variability in the duration of operations which normally occurs in a computer. For example, the time a processor takes to perform a multiplication often depends on the size of the numbers being multiplied. In functional languages there are even more problems due to storage management which could affect the duration of any operation. Further, on any system which allows several computations to timeshare a single processor there is scheduling to consider. One possible solution could be to give a maximum duration for each operation which allows for all these factors. However this implies that all occurrences of this operation will take that maximum time; these operation durations could be very large giving upper bounds which are too weak for real-time programming.

Both LUSTRE and ART make assumptions about the real-time properties of their implementations, assumptions that are probably unrealistic. This approach is in contrast to that of languages like Ada which assume no real-time properties of implementations. Instead, in line with the defensive programming philosophy, they provide primitives to allow programs to monitor their own progress in time by referencing a real-time clock. In Ada this is done via the CLOCK primitive supplied by the CALENDAR package. In the implicit time determinate language PFL [Holmstrom 83] the value of current time can be obtained by evaluating the primitive **`time`** which takes no arguments and returns a representation of the current time in seconds. However, since **`time`** takes no arguments but will return a different result each time it is evaluated it is non referentially transparent, although it can be given a fully determinate definition using herring-bone domains.

2.4.3 Real-time clocks in functional programming languages

A referentially transparent way of providing access to a real-time clock in a functional language, suggested in [Burton 88], is to treat clock values as a lazily evaluated input stream. Whenever the program wishes to know the time it references the head of the clock stream. The system then instantiates the head of the clock to the current time. Any subsequent reference to the head of the clock will obtain the same value so this preserves referential transparency. For example, the program below waits until `1000` time units after the start of execution before echoing its input.

```
output = wait(input,1000,clock)                                (2.32)
where
wait = lambda (i,t,clock).
         If t ≤ Head(clock)
         Then i
         Else wait(i,t,Tail(clock)) ;
endwhere
```

Note that after it is used each `clock` value is discarded, by taking `clock`'s tail, since it no longer denotes the current value of time.

In the last section of this chapter we shall bring together what we have learned about different approaches to real-time programming. From these alternative approaches, and from what we have discovered in this section about the remaining problems in applying functional languages to real-time programming, we shall select the basic characteristics of the language **Ruth**.

2.5 Conclusion

In this chapter we have examined the time expressibility offered by a range of real-time languages. Broadly we can identify two types of approach : the pragmatic, defensive approach and the theoretical, optimistic approach.

The defensive languages, typified by Ada, assume nothing about the real-time behaviour of a particular language implementation and give no guarantees that specified deadlines will be met. When working with such languages programmers must program in a very defensive manner : if there is the slightest doubt as to whether software will meet its constraints then it is rewritten. Defensive languages provide facilities to allow software to monitor its own progress in time so as to detect, and recover from, failures to meet deadlines. This usually involves writing a large amount of error handling code which (hopefully) is never executed. Most, if not all, languages used industrially for real-time work are defensive in nature.

The optimistic approach, typified by LUSTRE, assumes a particular real-time behaviour from language implementations. In LUSTRE's case the assumption is that of strong synchrony : that machine operations take negligible time and thus specified deadlines will always be met. LUSTRE thus provides no support for the detection of, and recovery from, failures to meet deadlines since such failures are assumed not to occur. Programs written in LUSTRE concentrate upon specifying the temporal behaviour required rather than on how that behaviour is to be achieved.

One of the advantages often cited for functional languages is that they allow the programmer to concentrate upon specifying what result is required rather than how that result is to be achieved; at first glance LUSTRE extends that benefit to temporal behaviour. However, how to produce the required result automatically from a functional program is a well understood problem; to guarantee the temporal behaviour implicit in a LUSTRE program is not. A programmer could write programs for which correct temporal behaviour requires that the implementation be able to perform, for example, multiplications in negligible time.

The other optimistic language we considered, ART, does not assume strong synchrony; instead the assumption is that the exact time taken for any machine operation is known. Thus temporal behaviour can be predicted from the semantics of the language. This is not as strong an assumption as strong synchrony but may be equally unrealistic due to the variability of duration of machine operations. Furthermore, proving the temporal behaviour of a program from the semantics of language is a complex task for all but the most trivial of programs, and is very error prone without substantial machine assistance and checking.

In the future machine performance may improve to such an extent that the strong synchrony hypothesis is viable for all but the narrowest of real-time problems. Equally, machine assisted program proving may also become possible. Until then we are forced onto the defensive : real-time programs must be able to monitor their own progress in time and must also be able to detect and recover from failures to meet deadlines.

The implicit time determinate functional languages examined in Section 2.4 modelled events as elements of infinite streams. This seems the obvious approach for functional languages and is the approach we shall follow in **Ruth**. To allow **Ruth** to specify when events are to occur, and to detect when events have occurred, we shall associate with each stream element a time value, or timestamp, and will refer to such timestamped streams as **channels**. By testing timestamp values a program can detect when events occur. This approach is similar to that taken by real-time Lucid save the channels have single valued timestamps rather than time windows. The use of single valued timestamps avoids implicit time determinance in the detection of when events occur. In common with all implicit time determinate operators, real-time Lucid's `time` primitive can be given a determinate definition via herring-bone domains; however, and once again in common with all implicit time determinate operators, its result cannot be predicted from the text of a program without a great deal of semantic analysis. Although using single values is less convenient than time windows we believe that easier understanding of a program's real-time behaviour more than compensates for this.

Partly for the same reason, but mostly because implicit time determinate operators like `amb` are not sufficiently discriminating for a real-time language, we shall extend our test on channel timestamps to detect when events have not occurred : if the timestamp of a message in a channel is not less than a certain number then the event did not occur by the time denoted by that number.

As a consequence of our adoption of the defensive approach **Ruth** will provide access to a real-time clock input stream to allow programs to monitor their progress in time and so detect and recover from failures to meet deadlines.

Once again it must be emphasised that **Ruth** is a language concerned mostly with time expressibility, software engineering issues having been tackled in some detail elsewhere. Many real-time systems are programmed as sets of parallel processes, both for reasons of program design and hardware structure. Rather than invest time and effort in examining the issues this raises for language design **Ruth** will directly borrow, and simplify, the occam [INMOS 84] model of a static set of processes communicating along point to point channels.

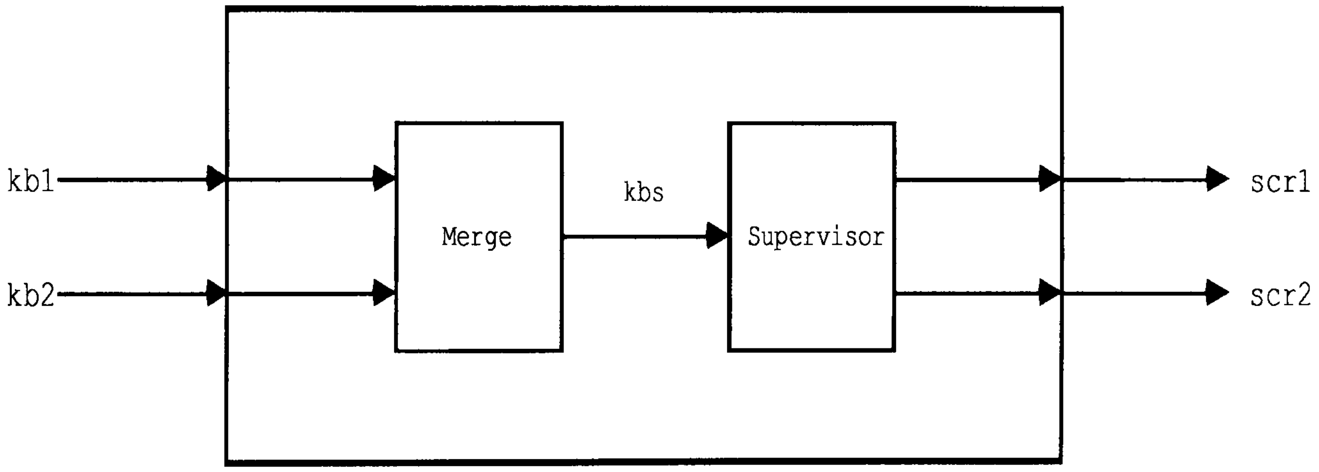
Chapter 3 : The Language Ruth

3.1 Introduction : Configurations, Processes And Channels

In this chapter we introduce the language **Ruth**. It is assumed that the reader is familiar with functional programming languages and thus the presentation will be fairly informal. The semantics of **Ruth** are outlined in the next two chapters and fully formally in Appendices 1 and 2. The complete syntactic definition of **Ruth** is given in Appendix 3.

The first step of the presentation is to introduce the model of the real world which **Ruth** assumes. We shall do this by informally specifying a real-time system and showing how it would be implemented as a set of independent, communicating **Ruth** processes. Consider the following system :-

System



System takes two keyboard inputs, kb1 and kb2, and produces output to two screens, scr1 and scr2. **System** comprises two processes : Merge and Supervisor. Process Merge takes kb1 and kb2 as input and merges them into kbs, on the basis of time of arrival of each message. Such timewise merges are fairly common in real-time systems, we have already given an implicit time determinate specification of one in (2.25). Supervisor takes kbs as input and echoes its contents to both scr1 and scr2. There is no timing restriction on Merge save that it execute as fast as possible, however, Supervisor must echo every message received on kbs to scr1 and scr2 within ten time units of its arrival at Supervisor. Whenever Supervisor receives a message which it cannot echo within this deadline it discards the message and sends 'time fault' upon scr1 and scr2. Whenever Supervisor has to wait for messages on kbs it sends 'waiting' upon scr1 and scr2.

A **Ruth** program is a set of processes executing in parallel and communicating via streams of timestamped messages, or channels. The set of processes and their configuration is fixed at compile time. **Ruth** does not allow processes to be dynamically created and/or destroyed at run time. To implement **System** in **Ruth** requires a configuration of two processes, Merge and Supervisor, communicating by the inter-process channel kbs. **System** also has four environment-process channels, kb1, kb2, scr1 and scr2, carrying the keyboard inputs and screen outputs. In **Ruth** this is expressed using the **Configuration** construct.

```
Configuration System                                         (3.1)
Output  scr1,scr2
Input   kb1, kb2
Is      scr1, scr2 = Supervisor (kbs) ;
        kbs        = Merge (kb1, kb2) ;
end.
```

A channel is an infinite stream of timestamped data values, or messages, each message denoting an event in the system. Each channel in a configuration must have a unique producing process, or be an input from the external environment, but may be consumed by any number of processes, including its producer, and/or the environment (though **System** does not contain an example of this). Each process defines a mapping from a tuple of input channels to a tuple of output channels. A process may have any number of input channels, including none at all; a process may also have any number of output channels, but must have at least one.

Channel timestamps are represented by non-negative integers. This seems natural in digital computers which can only represent discrete values accurately. We assume that the timestamps on channel messages represent values in real-time with an error of ±1/2 a tick. Provided the duration of a tick in discrete time is small enough we can treat timestamps as accurate representation of the continuous values. We shall assume that this is the case.

The timestamp of a channel message is interpreted as denoting the time at which the message will become available for use (i.e. will arrive) at its destination. Each message in the screen output channels `scr1` and `scr2` has a timestamp denoting when the message should appear on the screen, that is, when the event corresponding to the message should occur. The timestamps on the messages in `kb1` and `kb2` denote the time at which the keystroke was made, that is, when the keyboard event occurred.

A consequence of this interpretation of channel timestamps is that we must view the occurrence of the keystroke and the arrival of the message denoting that event at `Merge` as being simultaneous : channel communication is assumed to be infinitely fast. For an inter-process channel, particularly if the two processes are executing on the same physical processor, communication delays are likely to be small enough to ignore. However environment-process channels are likely to be carried by relatively long wires (e.g. of the order of a metre or more in an embedded system) and this will cause significant delay. One possible approach would be to add fixed amounts to the timestamps of environment-process channel messages to cope with the maximum possible delay. However, as with machine operations, such a maximum delay is difficult to specify and may cause large overheads if it is overestimated. In this work, for simplicity, we shall assume that all communication delays are negligible, whilst being aware that this is not entirely satisfactory and is an area requiring further work.

Note that although a message's timestamp denotes the earliest time at which its receiving process can use it there is no reason that the message will be used at that time. A process may choose to keep messages waiting until it is ready to deal with them, as, for example, `Supervisor` will do if `Merge` produces messages too quickly. We assume, however, that the environment never keeps messages waiting but reacts to them as soon as they arrive.

Given this interpretation of message timestamps the detection of events and when they occur simply requires a test on the value of a channel message's timestamp. However, as we shall see in Section 3.3, the detection of the non-occurrence of an event is not quite that straightforward, although basically, it is still a test on the timestamps of channel messages.

Returning to our example, `System`, we see that process `Merge` takes `kb1` and `kb2` as input channels and produces channel `kbs` as its output. In **Ruth** this is written

```
Process Merge                                            (3.2)
Input   kb1, kb2
Clock   c
Is      Expression
```

Unlike a configuration definition, a process definition does not name its output channels but does name a clock, in this case `c`. A unique clock is automatically supplied to each **Ruth** process at run-time, to provide real-time information, and the `Clock` keyword allows the programmer to provide a name by which the clock is referred to in `Expression`. `Expression` may be any **Ruth** expression and is called the **process expression**; it defines a mapping from the tuple of input channels listed after `Input`, and the named clock, to the tuple of output channels. In other words `Expression` defines a function from the input channels and the clock to the output channels : the **process function**. Of course, a `Configuration` or a `Process` definition is not a **Ruth** expression.

For simplicity we shall assume that there is no clock skewing between processes. If the processes are all executing on a single physical processor then they will all use the same physical clock and thus clock skewing cannot occur. If different physical processors are being used then skewing will almost certainly occur. [Lamport 78] gives an algorithm by which physical clocks on different processors can be synchronised, within certain fixed limits, by sending timestamped messages between the processors. The size of the limits depend on the frequency with which messages are exchanged and the communication delays involved. By using Lamport's algorithm, or something similar, it is possible to make clock skewing invisible to the **Ruth** programmer and consequently in the rest of this work we shall assume it does not occur.

In this section we have been concerned with laying the basic framework of a **Ruth** program as a static configuration of parallel processes communicating via channels, each process defining a function from its (possibly empty) tuple of input channels and clock input to its tuple of output channels. A channel is produced by exactly one process but may be consumed by any number of processes, including its producer. The timestamps on channel messages denote when the message will arrive at its destination, thus allowing the programmer to define when events are to occur and, by testing timestamp values, to determine when events did occur.

In the next section we shall briefly introduce the standard functional part of **Ruth**. Section 3.3 discusses channels and, in particular, the way in which **Ruth** programs detect the non-occurrence of an event. Section 3.4 concerns real-time clocks and how they are used. In Section 3.5 we show how **Ruth** can be used to solve real-time problems by completing the definition of the fairly simple `System`, given above.

3.2 The Standard Subset Of Ruth

The primitive, or atomic, data objects used in **Ruth** programs are strings, integers and booleans. Strings are delimited by `'` and `'`, for example `'This is a valid string'`. The usual arithmetic operations on integers are provided : addition (+), subtraction (-), multiplication (*), division (/) and modulus (\); and the usual testing operators : greater/less than (>), greater/less than or equal to (≥). Equality (=) and non equality (≠) are defined on all atomic objects. On booleans **Ruth** provides the usual operators, **And**, **Or** and **Not**.

Ruth also provides the traditional **If**...**Then**...**Else** construct, for example

```
If (a + b ≥ 27) Or (c \ d ≠ 4)                          (3.3)
Then 'OK'
Else 'Not OK'
```

Functions are denoted in **Ruth** by **lambda** abstractions

```
lambda (a, b, c) . a + b - c                             (3.4)
```

and function application is by juxtaposition as usual

```
(lambda (a, b, c) . a + b - c) (4, 5, 6)                 (3.5)
```

Identifiers can be associated with values via **where** definitions.

```
a + b + (a/b) where a = 27 ; b = 32 ; endwhere          (3.6)
```

where does not allow recursive definitions, for this purpose the **whererec** (where recursive) construct is provided.

```
fact(3)                                                  (3.7)
whererec
fact = lambda (n). If n = 0 Then 1 Else n * fact (n-1) ;
endwhererec
```

For data structuring purposes **Ruth** uses the LISP notion of the s-expression. Any atomic value (i.e. a string, integer or boolean) is an s-expression; the primitive operator **Cons** pairs together two s-expressions to produce a new s-expression and the primitive operators **Head** and **Tail** return the first and second elements of a **Cons**ed pair respectively.

```
Cons ('Hello', 5)   = ['Hello', 5]                        (3.8)
Head (['Hello', 5]) = 'Hello'
Tail (['Hello', 5]) = 5
```

The empty s-expression is denoted by the keyword **Nil** and the predicate **isNil** has the obvious meaning. A further predicate, **Atom** tests whether or not an s-expression is an atomic value. Note that in LISP **Nil** is an atom whereas in **Ruth** it is not.

```
isNil (5)            = false                              (3.9)
isNil (['Hello', 5]) = false
isNil (Nil)          = true

Atom (5)             = true
Atom (['Hello', 5])  = false
Atom (Nil)           = false
```

The semantics of **Ruth** given in Chapter 5 assume a normal order evaluation strategy such as is provided by the technique of lazy evaluation (e.g. [Henderson & Morris 76], [Friedman & Wise 76]). Consequently **whererec** can be used to define infinite data structures.

This completes our survey of the standard functional subset of **Ruth**. The assumption throughout this section has been that the reader is familiar enough with functional programming languages to require nothing more detailed.

In the next section we shall look at the first major step in providing **Ruth** with time expressibility : the introduction of timestamped streams of data values allowing **Ruth** programs both to determine when events took place and to define when events should take place.

3.3 Channels, Timestamps, And The `Ready` Test

In Section 3.1 we introduced the notion of a **channel** : an infinite stream of messages denoting events. The data value in a message defines what the event is and the timestamp when the event is to occur.

In **Ruth** the timestamps in channel messages are represented by non-negative integers and only atomic data objects are allowed as message data values. The major reason for this restriction is that of efficiency and simplicity of communication. Since the semantics of **Ruth** assume normal order evaluation data structures are only evaluated as far as is necessary to produce the immediately required result. Any s-expression may contain evaluated and unevaluated parts. The unevaluated parts are represented by, potentially very large, "recipes" for computing their values. If s-expressions are allowed as message data values an implementation must either traverse their whole structure and compute the values of all the recipes, or must transmit the recipes as part of the message. Either strategy would be a significant overhead, both in time and complexity. By restricting message data values to be atomic objects **Ruth** avoids this problem since an atomic object is either totally defined or totally undefined, it contains no recipes. If the atomic object is defined it is transmitted; if it is undefined the recipe representing it is forced and the resulting atomic object is transmitted.

We shall denote channel messages by enclosing them between "`{`" and "`}`"; for example `{t,a}` denotes the atom `a` stamped with the time `t`.

A channel is a head-strict, infinite, stream of messages. That is to say that **Ruth** insists that the first, or head, message in a channel must be completely evaluated to construct the channel so as to avoid the problems with partially evaluated messages outlined above. However the sending of the head message in a channel is totally unaffected by the contents of the rest, or tail, of the channel, and thus the tail need not be evaluated for the head message to be sent.

Channels will be delimited by "`[`" and "`]`", for example the channel whose first message is `{0,'Hello'}` and whose subsequent messages form the channel `rest` will be written

```
[{0,'Hello'}, rest]                                              (3.10)
```

There is one further important property of **Ruth** channels : the times denoted by the timestamps in channel messages must be strictly increasing. The interpretation of a channel is that it is an ordered sequence of messages, each of which denotes an event in the system and the implication of channels being head-strict objects is that their messages are produced in the order they occur in the channel. In other words we require that the first message in a channel denotes an event which occurs earlier than the events denoted by the messages in the remainder of the channel.

To ensure this we place an **incremental** interpretation upon channel timestamps. Let the first message in a channel be $\{t_1, a_1\}$, the second $\{t_2, a_2\}$, and so on. The time at which the event denoted by $\{t_1, a_1\}$ occurs is t_1; the time at which the event denoted by $\{t_2, a_2\}$ occurs is t_1+t_2+1. In general, if the time denoted by the timestamp of the n^{th} message is T_n then the time denoted by the timestamp of the $n+1^{th}$ message is $T_n+t_{n+1}+1$. Note that the +1 ensures that zero valued timestamps still denote a later time than their predecessor. For convenience, the **Ruth** programmer will use absolute time values for timestamps; the incremental interpretation of timestamps is simply a notational convenience when expressing the semantics of the language.

The data value part of the first message in a channel is referenced by the **Ruth** primitive **HeadCh** (*head channel*) and the **Time** primitive returns the timestamp of the first message, that is, when the event it denotes occurs. The tail of the channel is returned by the primitive **TailCh** (*tail channel*).

```
HeadCh ([{t,a}, rest]) = a                                       (3.11)
Time   ([{t,a}, rest]) = t

TailCh ( [{t,a}, [{t',a'}, rest'] ] ) = [{t+t'+1, a'}, rest']
```

Note the required adjustment to the timestamp of the tail of the channel made in the calculation of **TailCh**.

The **Ruth** channel constructor is called `ConsCh` (*cons channel*). `ConsCh` takes as arguments an atomic data value and a number, together defining a message, which it adds at the head of the channel which is its third argument.

```
ConsCh (a, t, ch)                                          (3.12)
    = [{t,a}, (t ≥ t' → ⊥, [{t'-t-1, a'}, rest] ) ]
      where
      [{t',a'}, rest] = ch,
```

Here ⊥ denotes the undefined value. The message to be added to `ch` must have a lower timestamp than that at the head of `ch` since the ordering of channel messages is identical to the ordering of the events they denote in time. If this is not the case then a channel containing only the new message results; `ch` is "ignored" since it is inconsistent with the new message. Note that the timestamp of the head message of `ch` is adjusted in line with the incremental interpretation of channel timestamps when the result channel is constructed.

There is one other important restriction on channel construction which `(3.12)` above does not mention. Since message timestamps denote the time at which the message must be available at its destination it is obviously an error to add a message with timestamp `t` to a channel at a real-time later than `t` as there is no way that the message can be delivered on time. **Ruth** programmers must program in a defensive way, by using the clock inputs to monitor programs' progress in time, so as to detect situations where this might occur and take corrective action.

Note that the possibility that a message might be computed with an out of date timestamp can only be expressed in a semantic model which allows the expression of timing information. `Consch` will be fully specified when the semantics of **Ruth** are given in Chapter 5.

In Section 3.1 we commented that **Ruth** processes map tuples of input channels into tuples of output channels. In **Ruth** tuples are constructed by listing their component channels between "`{`" and "`}`" and channels are selected from tuples via the `!` operator, overleaf.

```
{ch₁, ... , chₙ} ! m                                          (3.13)
    = (1 ≤ m ≤ n → chₘ, ⊥)
```

Note that the channels in a tuple are numbered from `1`.

We have now introduced enough of **Ruth** to allow us to write programs which can define when events are to occur and respond to when events do occur. For example, the process `Merge` from Section 3.1 requires that two input channels be merged into one output channel on the basis of when the input messages arrive : a timewise merge. A **Ruth** function for such a merge is given below.

```
merge                                                         (3.14)
   = lambda (ch1, ch2).
       If Time(ch1) ≤ Time(ch2)
       Then ConsCh(HeadCh(ch1), Time(ch1)+d,
                    merge(TailCh(ch1), ch2) )
       Else ConsCh(HeadCh(ch2), Time(ch2)+d,
                    merge(ch1, TailCh(ch2)) )
       where
       d = ... ;  -- a non-negative integer constant
       endwhere
```

In **Ruth** "--" denotes the start of a comment and comments are terminated by the end of the line. Each time a message is produced by `merge` its timestamp is increased by `d` so that `merge` must process messages within `d` time units of their arrival to avoid channel construction errors.

The question is, will this definition of `merge` evaluate fast enough to meet this requirement? The answer is almost certainly not. The expression

```
Time(ch1) ≤ Time(ch2)                                         (3.15)
```

cannot be evaluated unless the first message in both `ch1` and `ch2` is available. However all we are trying to determine is which of the two channels produces a message first. If `ch1` has produced a message and `ch2` has not then `(3.15)` is obviously `true` and we need not wait for `ch2`. In many situations we certainly would not want to wait for `ch2` to produce a message, for example, if `ch1` carried messages from a smoke detector and `ch2` carried messages from a keyboard. It highly likely that if a smoke detector has sensed the presence of smoke it will be a considerable time before another key is pressed on the keyboard. We would wish to pass on the smoke detector signal immediately and not have to wait for a keyboard input. Having to wait for both channels to produce a message before deciding between them will almost certainly ensure that the delay of `d` time units will be exceeded.

In fact, timewise merging is just a specific case of one of our requirements for a real-time language : the ability to detect that an event has not occurred. Having to wait until an event does occur in order to detect that it did not occur by a particular time is clearly undesirable; a check for an event at time `t` should produce a result at, or as close as possible, to time `t`. Since **Ruth** timestamps denote the actual times that events occur such a test is trivial to define : the **Ready** test.

```
Ready ([{t,a},rest], n) = t ≤ n                              (3.16)
```

The advantage of using **Ready** over using the **Time** function is that we can rely on the timestamps denoting the actual real-time at which the channel's messages become available to the **Ready**. An evaluation of the expression **Ready**`([{t,a},rest],n)` can return a result immediately if the channel `ch` already has a message available. If there is no message currently available in `ch` then evaluation need only be suspended till after the time denoted by `n`; after `n` has passed there is no way that a message with a timestamp less than or equal to `n` can appear in `ch` : the **Ready** test evaluates to `false`. Note that this timeout ability can only be defined in a semantic model containing timing information and this is done in Chapter 5.

Using **Ready** we can rewrite `merge` as shown overleaf.

```
merge                                                          (3.17)
   = lambda (ch1, ch2, n).
        If Ready(ch1,n)
        Then If Ready(ch2,n)
             Then If Time(ch1) ≤ Time(ch2) Then ans1 Else ans2
             Else ans1
        Else If Ready(ch2,n) Then ans2 Else merge(ch1, ch2, n+k)
        where
        ans1 = ConsCh(HeadCh(ch1), Time(ch1)+d,
                      merge(TailCh(ch1), ch2, n+k) ) ;
        ans2 = ConsCh(HeadCh(ch2), Time(ch2)+d,
                      merge(ch1, TailCh(ch2), n+k) ) ;
        d = ... ;  -- a non-negative integer constant
        k = ... ;  -- a non-negative integer constant
        endwhere
```

Here `k` is a constant determining the *sampling rate* of `merge`, that is, how often `merge` checks for messages. If the sampling rate is too slow (i.e. `k > d`) then we will certainly violate the timing requirement by keeping a message waiting for longer than `d` time units. However the sampling rate must not be faster than the underlying implementation of `merge` can cope with. If each evaluation of `merge` takes `i` time units and `i > k` then the value of `n` is going to fall further and further behind the current time and the timing requirement will eventually be violated. Thus, the correct value for `k` is in the range `i` to `d`. Obviously, in the case where `d < i` there is no possibility of satisfying the timing requirement. The value of `d` is known but determining that of `i` is more problematical, given the problems of predicting the exact timing behaviour of a computer.

Although, without knowing the value of `i`, we cannot justify ourselves formally, we can pragmatically argue that our new version of `merge` has a much better chance of satisfying the timing requirement since it will not waste time waiting for unavailable messages, and thus its value for `i` will be lower.

In many cases we require that the program be able to detect at run time if it is likely to fail to meet its deadlines and take remedial action. This is only possible if the program knows what the current time is. Channels are little help in this: although a channel timestamp denotes when a message arrives at its destination it does not denote when the message is finally referenced. A process may leave its inputs queued for some time before referencing them, so that the timestamps will bear little connection to the current time. In the next section we consider the clock inputs which are supplied to **Ruth** programs and show how they can be used to provide reference to the real-time.

3.4 Clocks And "Real" Real-Time

In the semantics given in Chapter 5 every **Ruth** program is supplied with a tree of time values (or clock) as suggested in [Burton 88] and each **Ruth** process is given a different sub-tree of the clock so that each process has a unique notion of the current time. The only reason for using a tree in preference to a linear list for clocks is so that unique sub-trees can be easily extracted in the semantics to allow for time independent evaluation of sub-expressions; operationally a process will simply build a list of values read from its processor's physical clock and thus to the **Ruth** programmer a clock is just a list of time values.

A clock tree is composed of a node holding a non-negative integer denoting the current time and two sub-trees containing the times of future events. As the tree is (lazily) evaluated each of the nodes is instantiated with the value of system time *at the time at which the node is instantiated*, thus giving programs reference to the current time. We shall denote a clock as a triple enclosed between "[" and "]", for example [t,l,r] denotes the clock with node value t and sub-trees l and r.

Since clock nodes will be instantiated in order (i.e. parent node, then child nodes) the values held in the child trees' nodes must denote times later than the parent's node value. To ensure this clock node values are given the same **incremental** interpretation as was given to channel timestamps. The number in the root node of a clock tree, say t, denotes the current time; the times in the roots of its immediate children, say t_l and t_r, denote the times $t+t_l+1$ and $t+t_r+1$ respectively. In general, if the time denoted by node in a clock tree is T and the node values of its immediate child trees are t_l and t_r, then the times denoted by the child node values are $T+t_l+1$ and $T+t_r+1$ respectively. As with channels, the **Ruth** programmer will deal only with absolute time values, the incremental representation of clock times is merely a notational convenience.

The **Ruth** primitives for accessing clock trees are the functions **HeadClk** and **TailClk** which are defined overleaf. (The names were chosen to emphasise the point that to a **Ruth** expression a clock is just a list of time values).

```
HeadClk ([t, l, r]) =   t                                    (3.18)
TailClk ([t, l, r]) =   [t+t_l+1, l_l, r_l]
                        where
                        [t_l, l_l, r_l] = l
```

TailClk simply returns the left child of the clock tree, so treating the clock as a simple list. There is no **Ruth** primitive for constructing clock trees : they are purely inputs to the program, the programmer cannot build his own. Obviously the programmer should be careful that HeadClk is not supplied with a clock which has already had its node value instantiated since, if that were the case, the value returned by HeadClk would not represent the current real-time.

With HeadClk and TailClk we can write functions which explicitly respond to the passage of time. In the last section we gave a version of merge which scanned its input channels every k time units. We pointed out that determining what the value of k should be is difficult : too large and messages will be kept waiting too long; too small and merge would not be able to execute fast enough. By using the clock input to return the current value of time we can remove the need for k altogether and simply check the input channels "now".

```
merge                                                           (3.19)
  = lambda (ch1, ch2, clk).
      If Ready(ch1,now)
      Then If Ready(ch2,now)
           Then If Time(ch1) ≤ Time(ch2) Then ans1 Else ans2
           Else ans1
      Else If Ready(ch2,now) Then ans2 Else merge(ch1,ch2,tclk)
      where
      ans1 = ConsCh(HeadCh(ch1), Time(ch1)+d,
                    merge(TailCh(ch1), ch2, tclk) ) ;
      ans2 = ConsCh(HeadCh(ch2), Time(ch2)+d,
                    merge(ch1, TailCh(ch2), tclk) ) ;
      d    = ... ;  -- a non-negative integer constant
      now  = HeadClk(clk) ;
      tclk = TailClk(clk) ;
      endwhere
```

By using the clock input clk to provide values for the current time this version of merge is effectively parametrised with its speed of execution. This is a better implementation of our original specification for the Merge process from Section 3.1 in which we stated that the only timing requirement for Merge was that it execute "as fast as possible". However merge has a stronger timing requirement than executing as fast as it possibly can : it cannot delay an input message by more than d time units or a channel construction error will be produced. That restriction can be easily removed to produce a merge function that exactly meets the specification for process Merge.

```
merge                                                                  (3.20)
  = lambda (ch1, ch2, clk).
      If Ready(ch1,now)
      Then If Ready(ch2,now)
           Then If Time(ch1) ≤ Time(ch2) Then ans1 Else ans2
           Else ans1
      Else If Ready(ch2,now) Then ans2 Else merge(ch1,ch2,tclk)
      where
      ans1 = ConsCh(HeadCh(ch1), now + delta,
                    merge(TailCh(ch1), ch2, tclk) ) ;
      ans2 = ConsCh(HeadCh(ch2), now + delta,
                    merge(ch1, TailCh(ch2), tclk) ) ;
      delta = ... ;  -- a non-negative integer constant
      now   = HeadClk(clk) ;
      tclk  = TailClk(clk) ;
      endwhere
```

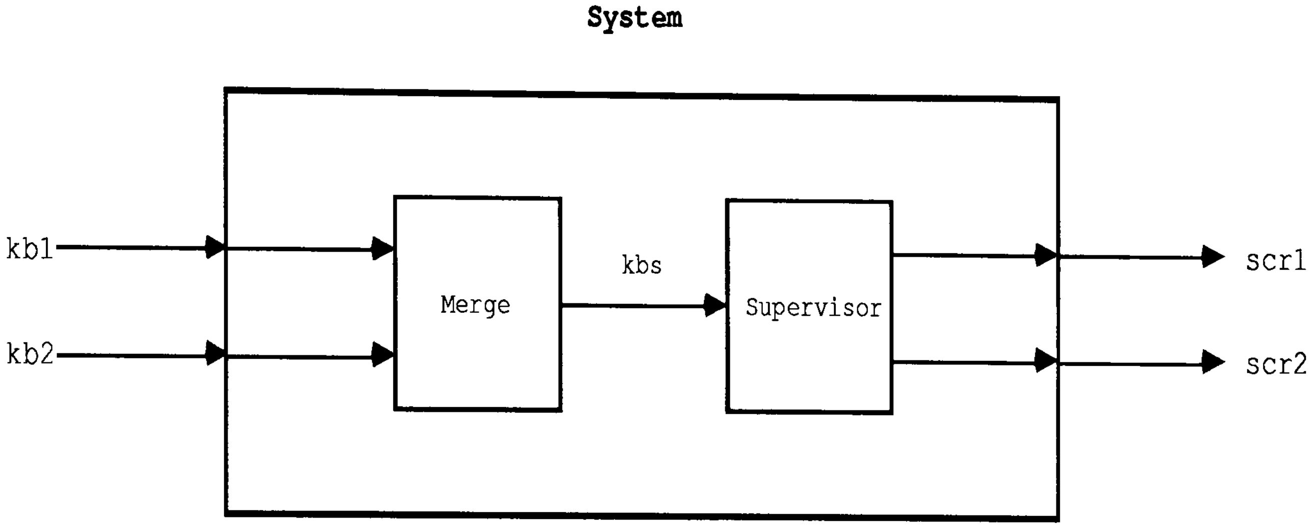
In (3.19) output messages were timestamped with the time they arrived at merge plus d. Here output messages are timestamped with the current time plus delta. It would be tempting to timestamp output messages with the current time alone but this would be an error. The clock must be instantiated to produce the timestamp before the message can be sent and thus, even assuming no transmission delays, the timestamp must be out of date before the message arrives at its destination. The delta value is used to allow for the unavoidable delay between channel construction and message arrival.

In real-time programs it is a common operation to timewise merge two or more channels and apply some function to the resulting channel. In the case of (3.20) the function applied is the identity function. It would be useful to include a higher-order function which would take as arguments a tuple of channels and a function and would apply the function to the timewise merge of the channels. This is only one example of the way in which higher-order functions, in this context often called I/O combinators, can be used to aid the writing of clear, concise functional programs. For further details see [Thompson 86], [Bird & Wadler 88] and [Jones & Sinclair 89].

This completes the informal description of **Ruth**. In the next section we shall complete our implementation of `System` from Section 3.1. This will serve two purposes : to show that **Ruth** can be used to solve real-time problems, even though the problem itself is somewhat trivial; and, more importantly, to further illuminate the language itself.

3.5 A Worked Example

In Section 3.1 we gave an informal specification of the simple real-time system pictured below.



The purpose of this system is to send all messages received from `kb1` and `kb2` to both `scr1` and `scr2`. Messages are to be sent to `scr1` and `scr2` in the order they arrived at `kb1` and `kb2`, to ensure which they are timewise merged by the process `Merge`. The only timing restriction on `Merge` is that it process messages as fast as possible. Process `Supervisor` takes the merged channel produced by `Merge` and sends it to both `scr1` and `scr2`. The timing restriction upon `Supervisor` is that it must pass on all the messages on `kbs` within ten time units of their arrival and any message which cannot be output within this deadline results in the message `'time fault'` being output. There is no restriction on how often `Supervisor` checks for messages in `kbs`, but whenever a check is made and no message is present the message `'waiting'` must be output.

The skeleton **Ruth** program to implement `System` is as follows

```
Process Merge      Input kb1, kb2 Clock clk1 Is ... ;      (3.21)
Process Supervisor Input kbs      Clock clk2 Is ... ;

Configuration System
Output  scr1,scr2
Input   kb1, kb2
Is      scr1, scr2 = Supervisor (kbs) ;
        kbs        = Merge (kb1, kb2) ;
end.
```

We first define the process and then the way in which they are configured to form **System**. In (3.20) we gave a definition of a timewise merge of the form required for process Merge. Thus the complete definition of Merge is

```
Process Merge                                                  (3.22)
Input kb1, kb2
Clock clk
Is  { merge (kb1, kb2, clk) }
    whererec
    merge = lambda (ch1, ch2, clk) ... ; -- As (3.20)
    endwhererec
```

Notice that the call of merge function is enclosed between "{" and "}" thus embedding its result channel into a tuple. This is required because **Ruth** processes produce tuples as their results. Note also that **Ruth** is a case sensitive language : there is no confusion between the process Merge and the function merge.

The skeleton for Supervisor is

```
Process Supervisor                                             (3.23)
Input kbs
Clock clk
Is  { out, out }
    whererec
    out = supervisor(kbs, clk) ;
    endwhererec
```

Since Supervisor sends the same output channel to both scr1 and scr2 the result of the application of the function supervisor is simply given twice in the output tuple. The function supervisor is defined overleaf.

```
supervisor                                                          (3.24)
 = lambda (kbs, clk).
     If Ready(kbs, now)
     Then If next_out - 10 > Time(kbs)
          Then ConsCh('time fault', next_out, rest)
          Else ConsCh(HeadCh(kbs), next_out, rest)
     Else ConsCh('waiting', next_out, supervisor(kbs, tclk))
     whererec
     now      = HeadClk(clk) ;
     tclk     = TailClk(clk) ;
     next_out = now + delta ;
     delta    = ... ;  -- a non-negative integer constant
     rest     = supervisor(TailCh(kbs), tclk) ;
     tclk     = TailClk(clk) ;
     endwhererec
```

Note the use of delta to allow for the delay between constructing the channel and the message being sent. If the first message on the channel arrives before next_out - 10, Supervisor cannot meet its deadline and the 'time fault' message will be sent instead.

As suggested by its name, Supervisor acts as a simple output supervisor for **System**, detecting, and recovering from, timing errors caused either by the lack of input from the keyboards ('waiting') or through its own failure to execute quickly enough ('time fault'). More complex strategies could easily be defined. For example, in certain cases a large backlog of messages may build up on kbs because Merge is producing messages faster than Supervisor can consume them, and sending a time fault message for each message in the queue which could not be delivered in time could cause Supervisor to fall further and further behind. A more sophisticated approach would be to remove all messages from the queue which arrived before next_out - 10, since these messages can definitely not be delivered in time, and only produce one 'time fault' message. The function discard removes all messages in a channel with timestamps less than, or equal to, a given value. discard is defined overleaf.

```
discard                                                 (3.25)
 = lambda (ch, t).
     If Ready(ch, t)
     Then discard(TailCh(ch), t)
     Else ch
```

To implement our more sophisticated recovery strategy we simply replace the `Then` branch of the inner `If...Then...Else` of `(3.24)`, which tests whether a message can be output within the deadline, by

```
ConsCh('time fault', next_out,                          (3.26)
        supervisor(discard(kbs,next_out - 10), tclk))
```

More complicated systems could easily be defined. For example, we might extend our requirement that `Supervisor` delay messages by no more than ten time units to the whole of **`System`**. This would involve using the `merge` function from `(3.19)`, and allowing, say, five time units for `Merge` and five for `Supervisor`.

The purpose of this section was to show how **Ruth** programs can be used to implement real-time systems. Although the problem chosen was a very simple one it does exhibit the essential feature of a real-time system : the necessity to meet deadlines. A more complex problem, an interactive computer game, is the subject of Chapter 5.

3.6 Conclusion

A **Ruth** program is a static configuration of processes communicating with each other, and with the external environment, via infinite streams of timestamped messages, or channels. A process is a function from a tuple of input channels and a real-time clock to a tuple of output channels. Each channel in a configuration is produced by a single process, or by the external environment, but may be consumed by any number of processes, including its producing process, and/or the external environment. A channel message denotes an event in the real-time system; its data value denotes what the event is and its timestamp when the event is to occur.

Ruth programmers can define when events are to happen by setting the timestamps in messages, and can detect when events occurred by checking timestamp values. However, checking message timestamps requires that a message be available to check, and so detecting that an event has not occurred cannot be done by simply checking a message's timestamp since, until the event does occur, the message will not exist. **Ruth** provides a primitive which will timeout messages : the `Ready` test. Although `Ready` appears to the **Ruth** programmer to be a simple test on message timestamps it relies upon the fact that message timestamps define when messages become available to allow it to return `false` if a message does not arrive by the specified time. In this way **Ruth** programs can also detect the non-occurrence of an event as well as its occurrence.

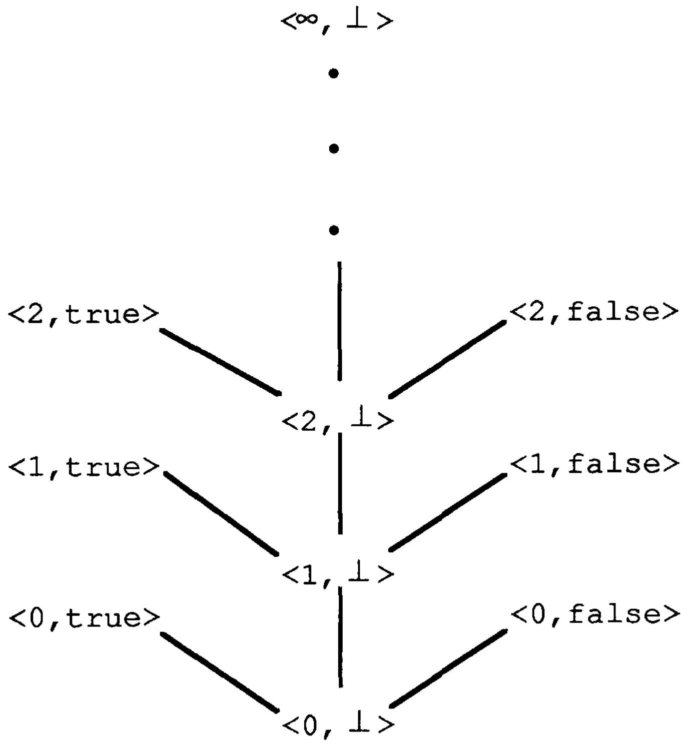
The remaining requirement is for a formal semantics allowing reasoning and transformation with **Ruth** programs. In the next chapter we construct a denotational semantics for **Ruth** based on the herring-bone domains introduced in Chapter 2.

Chapter 4 : Semantic Domains For Real-Time Programming

4.1 Introduction

It was noted in the last chapter that to define the semantics of **Ruth** requires a semantic model which incorporates timing information. In Chapter 2 we saw just such a model : the denotational semantics based upon herring-bone domains used in [Broy 83] to specify the semantics of the language ART.

Just as timestamps can be used to allow programmers to specify timing information in programs they can be used to add timing information to a semantics. The example given in Chapter 2 was of a herring-bone domain of booleans which can be represented diagrammatically as



When discussing herring-bone domains we shall observe the following convention : given a domain D (for example BOOL) the herring-bone domain which is constructed from it will be written as $\mathbb{D}$ (e.g. $\mathbb{BOOL}$). For the domain BOOL, the set of elements in the corresponding herring-bone domain $\mathbb{BOOL}$ is

$$\mathbb{BOOL} = \{<\mathtt{t},\mathtt{b}> \mid \mathtt{t} \in \mathrm{NUM}, \mathtt{b} \in \mathbf{BOOL}\} \cup \{<\infty,\perp>\} \qquad (4.1)$$

For the definition of $\mathbf{BOOL}$, the primitive domain of booleans, the reader is referred to Appendix 1. In (4.1) NUM is the set of non-negative integers which is used in preference to $\mathbf{NUM}$, the primitive domain of non negative integers, since it does not contain a $\perp$ element and this avoids the possibility of $\perp$ time values.

The ordering required on $\mathbb{BOOL}$, as can be seen from the diagram, is

$$\begin{aligned} &\forall\, t_1,t_2 \in \mathrm{NUM},\ b_1,b_2 \in \mathbf{BOOL} \ : && (4.2)\\ &\quad <t_1,b_1> \sqsubseteq <t_2,b_1> \Leftrightarrow (t_1 = t_2 \wedge b_1 \sqsubseteq_{\mathbf{BOOL}} b_2) \vee \\ &\qquad\qquad\qquad\qquad\qquad\quad (t_1 \leq t_2 \wedge b_1 = \perp) \\ &\text{and } <t_1,\perp> \sqsubseteq <\infty,\perp> \end{aligned}$$

A herring-bone domain, such as $\mathbb{BOOL}$, contains a chain of elements of the form

$$<0,\perp> \sqsubseteq <1,\perp> \sqsubseteq \ldots \sqsubseteq <t,\perp> \sqsubseteq \ldots \sqsubseteq <\infty,\perp> \qquad (4.3)$$

which, for obvious reasons, we shall refer to as its *spine*. Computation of an element of a herring-bone domain can be viewed as the production of increasing elements of the spine (i.e. undefined elements with greater and greater semantic timestamps) until the value becomes defined and the semantic timestamp becomes fixed, or, if computation of the value never terminates, the $<\infty,\perp>$ element eventually results. The interpretation of elements in a herring-bone domain is thus as follows :-

$<t,\perp>$ denotes that computation of the value has not been completed by time t.

$<t,v>$ where $v\neq\perp$ denotes that computation of the value v was completed at time t.

$<\infty,\perp>$ denotes that computation of the value is never completed.

In the ordering defined by (4.2) non-$\perp$ data values are incomparable unless they are paired with identical timestamps. On the surface this seems an unusual ordering; a more natural ordering might be thought to be that specified in (4.4) below

$$\begin{aligned} &\forall\, t_1,t_2 \in \mathrm{NUM},\ b_1,b_2 \in \mathbf{BOOL} \ : && (4.4)\\ &\quad <t_1,b_1> \sqsubseteq <t_2,b_1> \Leftrightarrow (t_1 \leq t_2 \wedge b_1 \sqsubseteq_{\mathbf{BOOL}} b_2) \\ &\text{and } <t_1,\perp> \sqsubseteq <\infty,\perp> \end{aligned}$$

but this is not the case. One of the major reasons for using herring-bone domains is to define the timeout properties of the **Ready** test. The ordering chosen for 𝔹𝕆𝕆𝕃 will carry over into all other herring-bone domains, including ℂℍ𝔸ℕ, the herring-bone domain of channels, which is defined later in this chapter. Assuming ℂℍ𝔸ℕ is ordered by analogy with `(4.4)` we have the following situation

a) **Ready** `(<10,[msg,rest]>, 10) = true` `(4.5)`
b) **Ready** `(<11,[msg,rest]>, 10) = false`

and, since `<10,[msg,rest]>` ⊑ `<11,[msg,rest]>`, for **Ready** to be monotonic requires that `true` ⊑ `false` which is clearly undesirable. As will be seen in Section 4.3 this is a somewhat simplistic picture of the semantics of the **Ready** test but the general principle holds : the ordering from `(4.4)` would make the **Ready** test non-monotonic.

Because of the required ordering, and because of the $<\infty,\perp>$ "top" element, 𝔹𝕆𝕆𝕃 cannot be directly constructed as a product of the NUM and BOOL domains. [Broy 83] makes no mention of how herring-bone domains might be constructed but by using lifted domains (e.g. [Cartwright & Donahue 82]) to model the semantic timestamps a definition can be given and this is done in the next section.

The purpose of this section was to introduce herring-bone domains and to outline some of their properties. The way in which herring-bone domains can be constructed is shown in the next section and the remaining sections of this chapter contain the definitions of the domains required to define the semantics of **Ruth**.

Once again it is assumed that the reader has a reasonable knowledge of denotational semantics and domain theory (see [Stoy 77], [Schmidt 86] for details) and so the approach will be fairly informal. This is to avoid obscuring the discussion with fine mathematical detail in the hope of more clearly exposing the underlying concepts. Appendix 1 contains the fully formal definitions of the domains shown in this chapter.

4.2 Herring-Bone Domain Construction

Before defining the domains to be used in giving the semantics of **Ruth** we must first show how herring-bone domains can be defined using the standard operators of domain theory. To illustrate the approach the definition of 𝔹𝕆𝕆𝕃 introduced in the last section will be given initially; the approach will then be generalised to any cpo.

The definition uses the standard domain operators ⊕ (coalesced sum) and ⊥ (domain lifting) which are defined in Appendix 1. The following constructors and selectors are used on sum domains.

The coalesced sum of two domains **A** and **B** is written $\mathbf{A} \oplus \mathbf{B}$ (4.6)

Constructors :

$$\begin{array}{lcll} \bot & : & \to \mathbf{A} \oplus \mathbf{B} & \\ \mathtt{in}\mathbf{A} & : \mathbf{A} & \to \mathbf{A} \oplus \mathbf{B} & \text{where } \mathtt{in}\mathbf{A}(\bot_{\mathbf{A}}) = \bot \\ \mathtt{in}\mathbf{B} & : \mathbf{B} & \to \mathbf{A} \oplus \mathbf{B} & \text{where } \mathtt{in}\mathbf{B}(\bot_{\mathbf{B}}) = \bot \end{array}$$

Selectors :

$$(\mathtt{Cases}\ x\ \mathtt{of}\ \mathtt{is}\mathbf{A}(a) \to e_1,\ \mathtt{is}\mathbf{B}(b) \to e_2\ )$$

such that

$$(\mathtt{Cases}\ \mathtt{in}\mathbf{A}(a)\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \to e_1,\ \mathtt{is}\mathbf{B}(y) \to e_2)$$
$$= (\lambda(x).e_1)\ (a)$$
$$(\mathtt{Cases}\ \mathtt{in}\mathbf{B}(b)\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \to e_1,\ \mathtt{is}\mathbf{B}(y) \to e_2)$$
$$= (\lambda(y).e_2)\ (b)$$
$$(\mathtt{Cases}\ \bot\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \to e_1,\ \mathtt{is}\mathbf{B}(y) \to e_2)$$
$$= \bot$$

in**A**, in**B**, is**A** and is**B** are often called domain injection and projection functions; where it would cause no confusion they will be omitted from semantic equations. A further notational convenience is the introduction of an else clause to the Cases notion defined above.

$$(\mathtt{Cases}\ x\ \mathtt{of}\ \mathtt{is}\mathbf{A}(a) \to e_1,\ \mathtt{else} \to e_2) \qquad (4.7)$$

such that

$$(\mathtt{Cases}\ \mathtt{in}\mathbf{A}(a)\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \to e_1,\ \mathtt{else} \to e_2)$$
$$= (\lambda(x).e_1)\ (a)$$
$$(\mathtt{Cases}\ \mathtt{in}\mathbf{B}(b)\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \to e_1,\ \mathtt{else} \to e_2)$$
$$= e_2$$
$$(\mathtt{Cases}\ \bot\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \to e_1,\ \mathtt{else} \to e_2)$$
$$= \bot$$

The constructors and selectors used with lifted domains are as follows

For any domain $\mathbf{A}$ the lifted domain is written $\mathbf{A}_\perp$ (4.8)

Constructor : $\perp \quad : \quad \rightarrow \mathbf{A}_\perp$

$\texttt{lift} \; : \mathbf{A} \rightarrow \mathbf{A}_\perp$

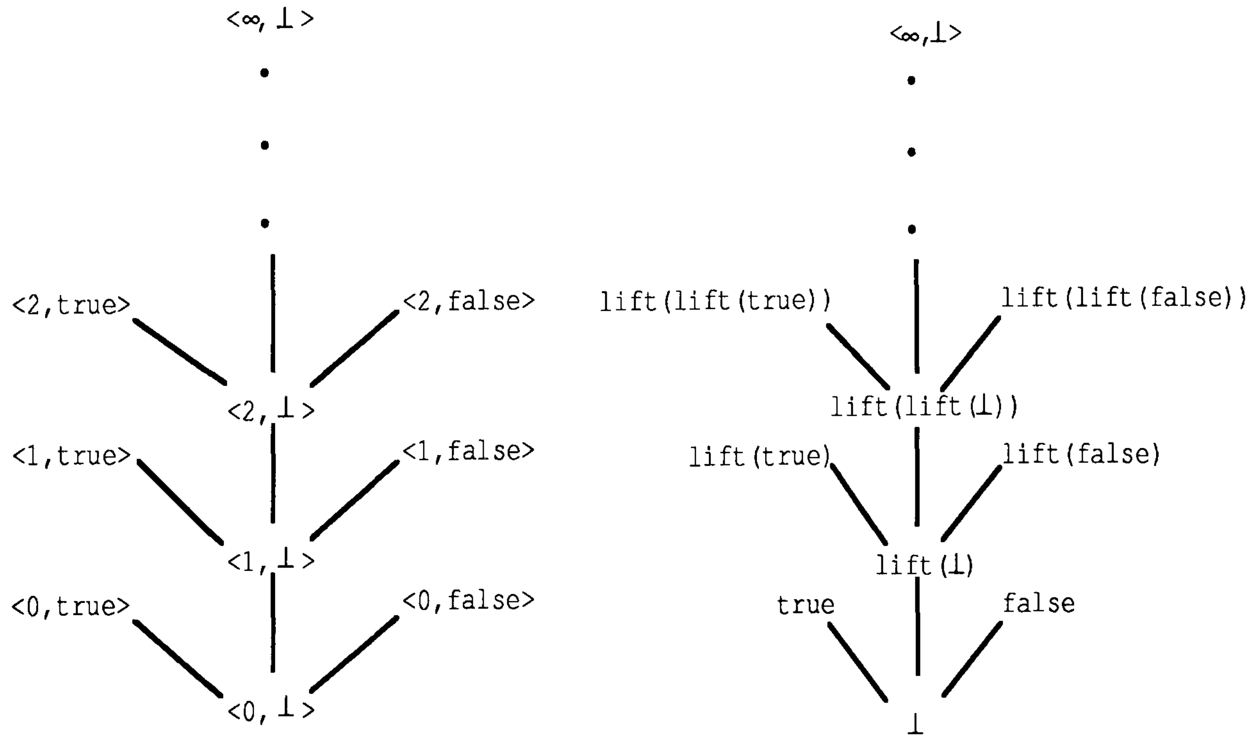
Selector : by pattern matching on `lift(a)` elements.

Having introduced the necessary notation the definition of $\mathbb{BOOL}$ can now be given.

$$\mathbb{BOOL} = \mathbf{BOOL} \oplus \mathbb{BOOL}_\perp \tag{4.9}$$

This definition does not directly mention semantic timestamps, instead they are modelled via the lifted domain $\mathbb{BOOL}_\perp$

<0,⊥>	is modelled by	⊥	(4.10)
<0,true>		inBOOL(true)	
<0,false>		inBOOL(false)	
<1,⊥>		in$\mathbb{BOOL}_\perp$(lift(⊥))	
<1,true>		in$\mathbb{BOOL}_\perp$(lift(inBOOL(true)))	
<1,false>		in$\mathbb{BOOL}_\perp$(lift(inBOOL(false)))	
<2,⊥>		in$\mathbb{BOOL}_\perp$(lift(in$\mathbb{BOOL}_\perp$(lift(⊥))))	
...		...	



The number of times an element of $\mathbb{BOOL}$ has been lifted into $\mathbb{BOOL}_\perp$ models the semantic timestamp to be associated with it. The "top" element $<\infty,\perp>$ is thus modelled by the limit of the spine elements of $\mathbb{BOOL}$: that is, by

$$\bigsqcup\{(\lambda(x).\mathtt{in}\mathbb{BOOL}_\perp(\mathtt{lift}(x)))^t\ (\perp)\ \mid\ t\geq 0\} \qquad (4.11)$$

where, for any function, f

$$f^0(x) = x$$

$$f^{t+1}(x) = f(f^t(x))$$

For reasons of space $<\infty,\perp>$ was used to represent this limit element in the diagram given in (4.10) above.

This isomorphism is proved in Appendix 1. The approach generalises in the obvious way to any non-recursively defined domain as shown below.

> For any domain expression E and domain $\mathbf{D}$ such that $\mathbf{D}$ is defined by $\mathbf{D} = E$ and E does not refer to $\mathbf{D}$, the herring-bone domain corresponding to $\mathbf{D}$ is defined by $\mathbb{D} = E \oplus \mathbb{D}_\perp$ (4.12)

Elements of domains defined as specified in (4.12) have exactly one semantic timestamp. When dealing with infinite structures like channels an infinite number of timestamps will be required, one for each message in the channel. In domain theory domains containing infinite and potentially infinite structures are defined by means of recursive definitions; for example the domain of head-strict infinite lists of non negative numbers could be defined

$$\mathbf{LIST} = \mathbf{NUM} \otimes \mathbf{LIST}_\perp \qquad (4.13)$$

Here $\otimes$ is the coalesced product domain operator which is defined in Appendix 1; elements of product domains will be denoted by listing their components between "[" and "]", for example [6,lift($\perp$)] is an element of $\mathbf{LIST}$. To form the herring-bone domain $\mathbb{LIST}$ a semantic timestamp must be included for each [number,list] pair in a list and this is achieved by the following definition

$$\mathbb{LIST} = (\mathbf{NUM} \otimes \mathbb{LIST}_\perp) \oplus \mathbb{LIST}_\perp \qquad (4.14)$$

Note that this definition has the same form and interpretation as (4.9) in that the semantic timestamps are modelled by the number of times a particular element has been lifted into $\mathbb{LIST}_\perp$. However the recursive use of $\mathbb{LIST}$ in $(\mathrm{NUM} \otimes \mathbb{LIST}_\perp)$ means that a semantic timestamp is associated with every [number,list] pair. Thus elements of $\mathbb{LIST}$ have the following structure :

$$<t_0,\ [n_0,\ \mathrm{lift}(<t_1, [n_1,\ \mathrm{lift}(<t_2, [n_2,\ \ldots]>)\ \ldots]>)\ ]> \qquad (4.15)$$

The use of ⊗ in the definition of $\mathbb{LIST}$ ensures that if the number is undefined then the whole list is undefined. However the tail of the list is lifted so that if it is the weakest element in $\mathbb{LIST}$, $<0,\perp>$, this does not cause the whole channel to be $<0,\perp>$. This is required for the head-strictness of lists : if the first element is undefined then the whole list is undefined; however, provided the first element is defined the list can always be constructed whatever the value of its tail. It is important that there should be no confusion between this use of domain lifting and its use on the right hand side of ⊕ to form the herring-bone domain.

Note that (4.14) is very similar to the definition of $\mathbb{CHAN}$, the domain of channels, given later in this chapter.

Generalising this to any recursive or non-recursive domain, a herring-bone domain can be constructed as follows :

> Given a domain definition $\mathrm{D} = \mathrm{F}(\mathrm{D})$, where $\mathrm{F}(\mathrm{D})$ is a domain expression which may or may not refer to D, the corresponding herring-bone domain $\mathbb{D}$ is defined $\mathbb{D} = \mathrm{F}(\mathbb{D}) \oplus \mathbb{D}_\perp$ (4.16)

(4.12) is simply a special case of (4.17). The constructors used for a herring-bone domain are

$$\begin{array}{lll} <\_,\ \_> & : \mathrm{NUM} \times \mathrm{F}(\mathbb{D}) & \rightarrow \mathbb{D} \\ <\infty,\ \perp> & : & \rightarrow \mathbb{D} \end{array} \qquad (4.17)$$

Of course (4.17) does not actually construct a herring-bone domain with numerical timestamps, but a domain which is isomorphic to this. The isomorphism was shown for the $\mathbb{BOOL}$ domain in (4.10) and (4.11) above, and is proved in Appendix 1; its essential feature is that the numeric timestamps in the herring-bone domain are modelled as injections into a lifted domain as follows :

$$
\begin{array}{ll}
\forall\ \mathtt{t} \in \mathtt{NUM},\ \mathtt{fd} \in \mathtt{F}(\mathbb{D}),\ \mathtt{fd} \neq \bot_{\mathtt{F}(\mathbb{D})}\ : & (4.18) \\
\quad \langle \mathtt{t},\mathtt{fd}\rangle = (\lambda(\mathtt{x}).\mathtt{in}\mathbb{D}_\bot(\mathtt{lift}(\mathtt{x})))^{\mathtt{t}}\ (\mathtt{inF}(\mathbb{D})\ (\mathtt{fd})) & \\
\quad \langle \mathtt{t},\bot\rangle\ \ = (\lambda(\mathtt{x}).\mathtt{in}\mathbb{D}_\bot(\mathtt{lift}(\mathtt{x})))^{\mathtt{t}}\ (\bot) & \\
\quad \langle \infty,\bot\rangle\ = \bigsqcup\{\ (\lambda(\mathtt{x}).\ \mathtt{in}\mathbb{D}_\bot(\mathtt{lift}(\mathtt{x})))^{\mathtt{t}}\ (\bot)\ \mid\ \mathtt{t} \geq 0\} & \\
\text{where, for any function } \mathtt{f} & \\
\quad \mathtt{f}^0(\mathtt{x})\ \ = \mathtt{x} & \\
\quad \mathtt{f}^{\mathtt{t}+1}(\mathtt{x}) = \mathtt{f}(\mathtt{f}^{\mathtt{t}}(\mathtt{x})) &
\end{array}
$$

Isomorphism is a strong enough property for us to assume that domains defined using (4.16) are actually the required herring-bone domains.

The selector used for herring-bone domains is as follows

$$
\begin{array}{ll}
(\mathbb{Match}\ \langle \mathtt{t},\mathtt{v}\rangle\ \mathbb{with}\ \langle \mathtt{t'},\bot\rangle \to \mathtt{e}_1,\ \langle \mathtt{t'},\mathtt{fd'}\rangle \to \mathtt{e}_2) & (4.19) \\
\text{such that} & \\
\quad (\mathbb{Match}\ \langle \mathtt{t},\bot\rangle\ \mathbb{with}\ \langle \mathtt{t'},\bot\rangle \to \mathtt{e}_1,\ \langle \mathtt{t'},\mathtt{fd'}\rangle \to \mathtt{e}_2) & \\
\quad\quad = (\lambda(\mathtt{t'}).\mathtt{e}_1)\ (\mathtt{t}) & \\
\quad (\mathbb{Match}\ \langle \mathtt{t},\mathtt{fd}\rangle\ \mathbb{with}\ \langle \mathtt{t'},\bot\rangle \to \mathtt{e}_1,\ \langle \mathtt{t'},\mathtt{fd'}\rangle \to \mathtt{e}_2) & \\
\quad\quad = (\lambda(\mathtt{t'},\mathtt{fd'}).\mathtt{e}_2)\ (\mathtt{t},\mathtt{d}) \quad \text{where } \mathtt{fd} \neq \bot &
\end{array}
$$

Although herring-bone domains are defined recursively as a sum of their basis domains (i.e. $\mathtt{F}(\mathbb{D})$) and the lifted herring-bone domain itself (i.e. $\mathbb{D}_\bot$), the constructors and selectors defined above still use the <t,d> notation introduced in Chapter 2. It is both more convenient and more intuitive to refer to the semantics timestamps directly as numbers rather than as a number of domain lifts and injections. $\langle\infty,\bot\rangle$ is used to construct the limit of the spine elements of a herring-bone domain since it is convenient to have a way of expressing non-termination directly. However, no case for $\langle\infty,\bot\rangle$ is required in the $\mathbb{Match}$...$\mathbb{with}$... selector since, by continuity, the result of applying the selector to $\langle\infty,\bot\rangle$ is the limit of the results of applying it to all the spine elements.

Note the potential for the $\mathbb{Match}$...$\mathbb{with}$... selector to be used in a non-monotonic fashion because it explicitly tests for the spine elements. The only place in which such a test is required is to define the timeout behaviour of **Ready** when it is applied to <t,⊥> channels; in all other cases it will be sufficient to map <t,⊥> elements to <t,⊥> as is done by the «...» notation defined below.

```
« <t,v> : <t',fd'> → e »                                       (4.20)
such that
    « <t,v> : <t',fd'> → e »
      = (Match <t,v> with
            <t',⊥>    →  <t',⊥>,
            <t',fd'>  →  (Match ( (λ(t',fd').e) (t',fd') ) with
                              <t",⊥>  → <max(t',t"),⊥>,
                              <t",v"> → <max(t',t"),v">
                          )
        )
```

$$\text{max} : \text{NUM} \times \text{NUM} \rightarrow \text{NUM}$$

$$\text{max} = \lambda\ (t_1,t_2).(t_1 > t_2 \rightarrow t_1,\ t_2)$$

Only the case for which the data value part of the herring-bone domain element is non-⊥ is given in the «...» notation, all <t,⊥> elements are mapped to <t,⊥>. Furthermore, the notation ensures that, for arbitrary v, the result for <t,v> is not weaker than that for <t,⊥>. Therefore the «...» notation cannot be used non-monotonically and so preserves the monotonicity of functions which use it to access herring-bone domains.

Where there is no possibility of confusion we shall abbreviate the «...» notation as follows :

```
« <t,v> → e »                                                  (4.21)
such that
    « <t,v> → e » = « <t,v> : <t,v> → e »
```

In essence, what the « ... » notation ensures is that the result of a function on an element of a herring-bone domain cannot be available until the element itself becomes available; in other words, that time cannot flow backwards. It is a pleasing attribute of herring-bone domains that monotonicity of functions can be interpreted in this way.

4.3 Definitions Of The Domains Required For The Semantics Of Ruth

4.3.1 $\mathbb{VAL}$: The domain of expressible values

The most important domain used in the semantic definition of **Ruth** is the herring-bone domain of expressible values, $\mathbb{VAL}$, since this domain contains all possible results of evaluating a **Ruth** expression.

$$\mathbb{VAL} = (\mathbf{S\text{-}EXP} \oplus \mathbf{FUNC} \oplus \mathbf{TUPLE} \oplus \mathbb{CLK} \oplus \mathbb{CHAN}) \oplus \mathbb{VAL}_{\perp} \tag{4.22}$$

Apart from the recursive reference to $\mathbb{VAL}_{\perp}$ required to construct the herring-bone domain $\mathbb{VAL}$ is the coalesced sum of the domains **S-EXP** (the domain of s-expressions), **FUNC** (functions), **TUPLE** (tuples), $\mathbb{CLK}$ (clocks) and $\mathbb{CHAN}$ (channels), each of which will be defined later in this section. Note that the domains of clocks and channels are themselves herring-bone domains. The semantic timestamps in $\mathbb{CLK}$ will be used to represent the time values in a clock and those in $\mathbb{CHAN}$ to represent the values of the user defined message timestamps in channels.

Note also that the use of the coalesced sum operator in (4.22) results in the weakest, $<0,\perp>$, elements of $\mathbb{CLK}$ and $\mathbb{CHAN}$ being identified with the weakest, $\perp$, element of **S-EXP** $\oplus$ **FUNC** $\oplus$ **TUPLE** $\oplus$ $\mathbb{CLK}$ $\oplus$ $\mathbb{CHAN}$. This could be seen as undesirable since even a zero semantic timestamp conveys some information which should not be lost, but in fact, as will be seen later, this is not the case.

4.3.2 **S-EXP** : The domain of s-expressions

The first domain to be defined is that of s-expressions. As was seen in Chapter 3 s-expressions are made up of atoms, that is, of booleans, integers and strings. Thus we have the domain of atoms, **ATOM**.

$$\mathbf{ATOM} = \mathbf{BOOL} \oplus \mathbf{INT} \oplus \mathbf{STRING} \tag{4.23}$$

The primitive domains of booleans (BOOL), integers (INT), and strings (STRING) are the usual, flat, cpos. A further component of the s-expression domain is the primitive cpo NIL which contains, besides ⊥, the element `Nil` which is used to denote the empty list. The definition of S-EXP is thus

$$\begin{array}{ll} \text{S-EXP} & = \text{NIL} \oplus \text{ATOM} \oplus \text{PAIR} \\ \text{PAIR} & = (\mathbb{S}\text{-}\mathbb{EXP} \times \mathbb{S}\text{-}\mathbb{EXP})_{\perp} \\ \mathbb{S}\text{-}\mathbb{EXP} & = \text{S-EXP} \oplus \mathbb{S}\text{-}\mathbb{EXP}_{\perp} \end{array} \qquad (4.24)$$

Here x is the product operator on domains and is defined in Appendix 1. The domain PAIR contains pairs of s-expressions constructed via the **Cons** primitive. Note that PAIR is actually composed of a pair of herring-boned s-expression domains. Note also that the pair is lifted to avoid identifying the pair $[<0,\perp>,<0,\perp>]$ with the ⊥ element of S-EXP. Both these actions are taken because s-expressions are lazily evaluated in **Ruth** : the elements of a **Cons**ed pair need not be evaluated in order for the **Cons** to be evaluated. Even if the elements of a **Cons**ed pair are undefined the **Cons** itself may be performed and thus $[<0,\perp>,<0,\perp>]$ is distinct from ⊥ in S-EXP. Further, since under a lazy strategy the elements of a **Cons** are not evaluated at the same time as the **Cons** itself, semantic timestamps are required for these elements; hence the herring-boned domain $\mathbb{S}\text{-}\mathbb{EXP}$ must be used.

4.3.3 FUNC : The domain of functions

The semantic timestamp added to elements of FUNC when they are embedded in $\mathbb{VAL}$ denotes when the function is available to be applied to its arguments (operationally, when the code corresponding to the function is loaded from memory). The function space domain F on elements of the expressible values domain $\mathbb{VAL}$ is

$$\text{F} = \mathbb{VAL} \rightarrow \mathbb{VAL} \qquad (4.25)$$

Note that F contains only single argument functions whilst **Ruth** allows any (finite) number of arguments to functions. This gives the following definition

$$\mathrm{F} = \mathbb{VAL} \rightarrow (\mathrm{F} \oplus \mathbb{VAL}) \qquad (4.26)$$

There is one further consideration. In **Ruth** both **lambda** expressions and process definitions produce elements of the function space. A function may be applied several times during the evaluation of a program, and thus at several different real times. This is represented in the semantics of **Ruth** by supplying a function with a clock argument from which it obtains the times at which its results become available.

$$\mathrm{FUNC} = \mathbb{CLK} \rightarrow \mathrm{F} \qquad (4.27)$$

When an element of FUNC is applied to a clock the result is an element of F "parameterised" with its evaluation time. This use of clocks is called *clock-driven* timing and will be discussed in some detail in the next chapter.

4.3.4 TUPLE : The domain of tuples

TUPLE is similar to the domain PAIR defined above in that it contains herring-bone sub-domains. Each channel in a tuple may be evaluated independently of any other, and of construction of the tuple itself, and consequently each channel requires a separate semantic timestamp.

$$\mathrm{TUPLE} = (\mathbb{ELEM}_\perp \otimes \mathrm{TUPLE}) \oplus \mathrm{NIL} \qquad (4.28)$$
$$\mathbb{ELEM} = \mathbb{CHAN} \oplus \mathbb{ELEM}_\perp$$

Note the use of the NIL domain to mark the end of a tuple. Also note that elements of the $\mathbb{ELEM}$ domain are identical in structure to elements of $\mathbb{CHAN}$ embedded in $\mathbb{VAL}$: they comprise an element of $\mathbb{CHAN}$ with an extra semantic timestamp added.

The use of ⊗ in the equation defining TUPLE ensures that tuples must have a finite length; however ⊗ has the effect of coalescing the least, $<0,\perp>$, elements of its argument domains into one least element in the TUPLE domain. We wish to distinguish between a tuple containing a $<0,\perp>$ member of $\mathbb{ELEM}$ and a totally undefined tuple and so each member of $\mathbb{ELEM}$ is lifted to add a new least element which can be coalesced into the least element of **TUPLE**.

Elements of the tuple domain will be constructed by listing them between "{" and "}" as below

$$\begin{array}{llll} \{\ \} & : & \rightarrow \mathbf{TUPLE} & (4.29) \\ \{\_,\ \ldots,\ \_\} & : (\mathbb{ELEM} \times \ldots \times \mathbb{ELEM}) & \rightarrow \mathbf{TUPLE} & \end{array}$$

such that

$$\{\ \} = \mathtt{in}\mathbf{NIL}(\mathtt{Nil})$$

$$\{el_1, el_2, \ldots, el_m\} = \mathtt{in}\mathbb{ELEM}_{\perp}\otimes\mathbf{TUPLE}\ ([\mathtt{lift}(el_1),\ \{el_2, \ldots, el_m\}])$$

Selection from elements of the tuple domain will be by pattern matching on $\{el_1, el_2, \ldots, el_m\}$ structures.

4.3.5 $\mathbb{CLK}$: The domain of clocks

There are two uses for $\mathbb{CLK}$ in the semantics of **Ruth**. Firstly, its elements are used to support clock-driven timing; secondly, $\mathbb{CLK}$ is embedded into $\mathbb{VAL}$ and its elements used to provide **Ruth** programmers with real-time clocks. We shall refer to these two types of clocks as semantic and program clocks respectively. In the last chapter it was assumed that a clock tree contained numbers. In fact clocks are defined using a herring-bone domain, the semantic timestamps being used to represent the real-time values.

$$\mathbb{CLK} = (\mathbb{CLK} \times \mathbb{CLK}) \oplus \mathbb{CLK}_{\perp} \qquad (4.30)$$

A clock of the form $<t_c, \perp>$ contains no information beyond that its next time value will be no less than t_c. Once the sub-clocks become defined, that is once the clock is of the form $<t_c, [l_c, r_c]>$, the value of t_c is fixed and denotes the first time value on the clock. Because of this interpretation of $\mathbb{CLK}$ elements the weakest clock, $<0, \perp>$, conveys no useful information. Thus, although the use of ⊕ in (4.30) means that the weakest element of $\mathbb{CLK} \times \mathbb{CLK}$, $[<0,\perp>, <0,\perp>]$, will be identified with the weakest element of $\mathbb{CLK}$, $<0, \perp>$, this loses no information.

When $\mathbb{CLK}$ is embedded into $\mathbb{VAL}$ to provide program clocks an extra semantic timestamp is added to each clock. Thus, a clock of the form $<t_c, [l_c, r_c]>$ becomes a program clock of the form $<t, <t_c, [l_c, r_c]> >$ when embedded into $\mathbb{VAL}$. It is important that the difference between the meanings of t and t_c is clear. The semantic timestamp on elements of $\mathbb{VAL}$, such as t, denote when values are computed; in the case of clocks when, for example, the result of a **TailClk** or an identifier reference to the clock is evaluated. The timestamps within clocks, such as t_c, denote actual values of real-time independent of when those values become available to the **Ruth** program. We shall refer to these values as clock timestamps and note that there need be no connection between clock timestamps and semantic timestamps, for example if a clock has already had its clock timestamps instantiated when it is accessed.

When a clock of the form $<t_c, \perp>$ is embedded into $\mathbb{VAL}$ it becomes a program clock of the form $<t, <t_c, \perp> >$. Since $<0, \perp>$ is the weakest element of $\mathbb{CLK}$ the use of $\oplus$ in the definition of $\mathbb{VAL}$ (4.22) causes $<0, \perp>$ to be identified with the weakest element of $\mathbf{S\text{-}EXP} \oplus \mathbf{FUNC} \oplus \mathbf{TUPLE} \oplus \mathbb{CLK} \oplus \mathbb{CHAN}$. Thus, when the $<0, \perp>$ clock is embedded into $\mathbb{VAL}$ to form a program clock a $<t, \perp>$ element of $\mathbb{VAL}$, for some semantic timestamp t, results. Once again however, since a $<0, \perp>$ clock contains no useful information this causes no problems.

A final point to note about clocks is that the incremental interpretation of clocks mentioned in the last chapter still holds. Given a clock tree $<t_c, [l_c, r_c]>$ the time denoted by its root t_c is t_c; the times denoted by the roots of l_c and r_c, say t_l and t_r are t_l+t_c+1 and t_r+t_c+1 respectively.

4.3.6 $\mathbb{CHAN}$: The domain of channels

A channel is a head-strict list of timestamped atomic values, or messages. The domain $\mathbb{CHAN}$ is thus defined (c.f. (4.14))

$$\mathbb{CHAN} = (\mathbf{ATOM} \otimes \mathbb{CHAN}_\perp) \oplus \mathbb{CHAN}_\perp \qquad (4.31)$$

The timestamps of messages in channels are supplied by the programmer in the **ConsCh** construct. In the herring-bone domain $\mathbb{CHAN}$ the message timestamps are represented by the semantic timestamps. Thus, where in Chapter 3 we wrote `[{10,'Hello'},rest]` for the channel containing the message `'Hello'` at (user defined) time `10`, followed by the channel `rest`, we now write `<10,['Hello',lift(rest)]>`.

As with clocks the incremental interpretation of channel timestamps mentioned in the last chapter still holds. Given a channel `<t,[a,lift(rest)]>` the actual time denoted by its first timestamp `t` is `t`; the time denoted by the first timestamp in `rest`, say $\mathtt{t}_\mathtt{r}$ is $\mathtt{t+t_r+1}$.

Also as with $\mathbb{CLK}$, when $\mathbb{CHAN}$ is embedded into $\mathbb{VAL}$ an extra timestamp is added. Thus the channel `<t,[a,lift(rest)]>` becomes $\mathtt{<t_v,<t,[a,lift(rest)]>>}$, and the channel $\mathtt{<t,\perp>}$ becomes $\mathtt{<t_v,<t,\perp>>}$. Once again it is important to be clear about the difference in the meanings of $\mathtt{t_v}$ and `t`. The semantic timestamp, $\mathtt{t_v}$, denotes the time at which the channel is computed, that is, the time at which the result of a **ConsCh**, **TailCh** or identifier reference to a channel is evaluated. The `t` values, one for each channel message, denote the user defined timestamps for each message, and will be referred to as message timestamps in the rest of this work.

The final point to note is the treatment of the $\mathtt{<0,\perp>}$ element of $\mathbb{CHAN}$ when $\mathbb{CHAN}$ is embedded into $\mathbb{VAL}$. Because of the use of ⊕ in `(4.22)` this element is identified with the ⊥ element of $\mathbf{S\text{-}EXP} \oplus \mathbf{FUNC} \oplus \mathbf{TUPLE} \oplus \mathbb{CLK} \oplus \mathbb{CHAN}$ and thus a $\mathtt{<t,\perp>}$ element of $\mathbb{VAL}$, for some semantic timestamp `t`, results. Just as with $\mathbb{CLK}$ however no useful information is lost; a $\mathtt{<0,\perp>}$ channel is a channel whose first message cannot have a timestamp of less than `0`, which is something which is trivially true for all channels.

4.3.7 ENV : The domain of environments

The final domain required is **ENV**, the domain of mappings from syntactic identifiers to elements of $\mathbb{VAL}$.

$$\mathbf{ENV} = \mathtt{Id} \rightarrow (\mathbf{UNDEF} \oplus \mathbb{VAL}) \qquad (4.32)$$

Here Id is the syntactic domain of identifiers and **UNDEF** is the primitive domain containing the elements $\{\perp, \mathtt{Undef}\}$ which is used to indicate that an identifier is not defined in a particular environment. The constructors and selectors for **ENV** are

Constructors : (4.33)

$$\begin{array}{lll}
\varnothing & : & \rightarrow \mathbf{ENV} \\
[\_ \rightarrow \_] & : \mathrm{Id} \times \mathbb{VAL} & \rightarrow \mathbf{ENV} \\
\_ \mathbin{\&} \_ & : \mathbf{ENV} \times \mathbf{ENV} & \rightarrow \mathbf{ENV}
\end{array}$$

Selector : $\_[\_] \ : \mathbf{ENV} \times \mathrm{Id} \rightarrow \mathbf{UNDEF} \oplus \mathbb{VAL}$

such that

$$\begin{array}{l}
\varnothing[\mathtt{I}] \\
\quad = \mathrm{in}\mathbf{UNDEF}(\mathtt{Undef}) \\
[\mathtt{I} \rightarrow \mathtt{v}]\ [\mathtt{I}'] \\
\quad = (\mathtt{I} = \mathtt{I}' \rightarrow \mathrm{in}\mathbb{VAL}(\mathtt{v}),\ \mathrm{in}\mathbf{UNDEF}(\mathtt{Undef})\ ) \\
(\rho_1 \mathbin{\&} \rho_2)\,[\mathtt{I}] \\
\quad = (\mathrm{Cases}\ \rho_2[\mathtt{I}]\ \mathrm{of} \\
\qquad\quad \mathrm{is}\mathbf{UNDEF}(\mathtt{Undef}) \quad \rightarrow \rho_1[\mathtt{I}] \\
\qquad\quad \mathrm{else} \qquad\qquad\qquad \rightarrow \rho_2[\mathtt{I}] \\
\qquad)
\end{array}$$

In (4.33) ρ is used to denote an environment and this convention will be followed in the rest of this work.

4.4 Conclusion

This chapter has concerned the semantic domains required to specify the language **Ruth**. The first section explored the elements and ordering required for herring-bone domains and explained the real-time interpretation of herring-bone domain elements. Section 4.2 showed how herring-bone domains could be constructed from any domain by a simple syntactic transformation of that domain's definition. Following this Section 4.3 gave the definitions of the particular domains required to specify the semantics of **Ruth**, both herring-bone and otherwise.

Having specified the domains required, the next chapter outlines the semantics of **Ruth**.

Chapter 5 : The Semantic Definition Of Ruth

5.1 Introduction

In this chapter we shall use the domains defined in Chapter 4 to outline the semantics of **Ruth**. Domain injection and projection functions have been omitted where this causes no confusion. The fully formal definition of **Ruth** can be found in Appendix 2.

The semantics of **Ruth** presented here will be based upon the herring-bone domains defined in the last chapter. Using herring-bone domains allows us to determine when values are evaluated since the information is contained in their timestamps. The first question to be answered is : how are the semantic timestamps to be paired with data values determined? In other words, assuming that the time at which the sub-expressions of an expression produced their results is known, when will the expression produce its result?

In Chapter 2 we saw two possible approaches to this problem. The strong synchrony hypothesis, used in LUSTRE, is an example of what we shall refer to as **data-driven** timing. Since all machine operations are assumed to take zero time the time at which an expression produces a result is totally determined by when the input data required by that expression becomes available. An alternative approach, which we shall call **delay-driven** timing, was used in specifying the semantics of the language ART : all machine operations are assumed to take a fixed, and specifiable, amount of time to perform.

Data-driven timing gives us a very simple abstraction from implementation details and makes the semantics easy to work with. Unfortunately it is not a very accurate model of the real world since it implies that computers are infinitely fast and so gives no real information about real-time behaviour.

On the other hand delay-driven timing requires that exact durations for every machine operation be specified and this can be difficult due to the variability of operation durations that usually occurs in a computer. It may be possible to put upper bounds on these durations but these upper bounds will usually be much larger than the average operation times, and, since any implementation will be constrained to take the specified amount of time to perform each operation, this will waste valuable computing time. Further, to give a delay-driven semantics to a language forces the specification of many low level details, for example the order of evaluation of function arguments, and, in a lazily evaluated language, when recipes are to be updated with with their values. Finally, the machine operation times specified in a delay-driven semantics will be implementation dependent : a particular delay-driven semantics will only apply to the language when it is executed by a particular implementation on a particular processor.

These problems with delay-driven timing are essentially caused by its *prescriptive* nature. They can be avoided by specifying a more *descriptive* semantics : instead of specifying exactly how long each machine operation is to take the semantics of **Ruth** will be parametrised with a clock from which the times at which expressions produce results will be read. This provides a higher level of abstraction than delay-driven timing since such things as evaluation orders need not be specified. Yet it allows the real world to be more accurately modelled than is possible with data-driven timing; by constraining the values read from the semantic clocks in certain ways we can express data dependency, for example that the result of an addition cannot be available until after the two operands are computed. We shall call this approach **clock-driven** timing.

For example, assume that the semantics are parametrised with the clock `c`. The logical `And` of two booleans could be defined as overleaf

$$\langle t_1,b_1\rangle\ \textbf{And}\ \langle t_2,b_2\rangle = \mathtt{after}(\langle \max(t_1,t_2),\ b_1 \wedge b_2\rangle,\ c) \tag{5.1}$$

```
after : $\mathbb{VAL} \times \mathbb{CLK} \rightarrow \mathbb{VAL}$
after
    = $\lambda(v,c)$ .
        « c : $\langle t_c, [l_c, r_c]\rangle$
            → « v : $\langle t,d\rangle$
                    → $(t_c > t \rightarrow \langle t_c, v\rangle,\ \mathtt{after}(\langle t,d\rangle, \mathtt{from}(l_c,t_c))$
              »
        »
```

```
from : $\mathbb{CLK} \times \mathrm{NUM} \rightarrow \mathbb{CLK}$
from = $\lambda(c,n)$ . « c : $\langle t_c, [l_c,r_c]\rangle \rightarrow \langle t_c+n+1, [l_c,r_c]\rangle$ »
```

The function `after` models the passage of time during execution of an operation which in a delay-driven semantics is achieved by simply adding a fixed amount to the value's semantic timestamp. When supplied with a `<semantic timestamp, data value>` pair `after` returns a pair comprising the data value with a semantic timestamp read from the clock but constrained to be strictly greater than its original semantic timestamp. We shall refer to this process as *ageing*. Note the use of the `from` function to take account of the incremental interpretation of clock values.

In the rest of this chapter we shall use herring-bone domains and clock-driven timing to specify the semantics of **Ruth**. In the next section we lay the foundations for this specification by defining some useful semantic functions. Section 5.3 outlines the semantics of the standard non real-time subset of **Ruth** and Section 5.4 covers the semantics of those parts of **Ruth** which have been added to cope with real-time systems : clocks, channels and, in particular, the **Ready** test.

5.2 Useful Semantic Functions

The semantics of **Ruth** expressions are defined using the evaluation function $\mathcal{E}_V$ which has the following signature

$$\mathcal{E}_V \; : \texttt{Exp} \to \mathbf{ENV} \to \mathbb{CLK} \to \mathbb{VAL} \tag{5.2}$$

Here `Exp` is the domain of syntactic expressions. $\mathcal{E}_V$ also takes as arguments an environment, and a clock to facilitate clock-driven timing. The environment holds identifier/value bindings and, as can be seen from `(4.33)`, if a particular identifier is not bound in an environment the `Undef` value results. Rather than deal with `Undef` values explicitly in the semantics the function `lookup` is used.

```
lookup : Id x ENV → 𝕍𝔸𝕃 (5.3)
lookup
    = λ(I,ρ) .
        (Cases ρ[I] of
          isUNDEF(Undef)  → <∞,⊥>
          is𝕍𝔸𝕃(<t,v>)    → <t,v>
        )
```

If the identifier is undefined in the environment the `<∞,⊥>` element results. Operationally, if an expression attempts to reference an undefined identifier then the expression will never produce a result : the program will crash.

When writing the semantic equations it will frequently be necessary to extract sub-clocks from the clock supplied to $\mathcal{E}_V$ to allow for the independent evaluation of sub-expressions. For this purpose the `extract` function will be used.

```
extract : ℂ𝕃𝕂 x NUM → ℂ𝕃𝕂                                      (5.4)
extract
    = λ(c,n) .
        « c : <t,[l,r]>
           → (n ≤ 0 → from(l,t), extract(from(r,t),n-1) )
        »
```

For the remainder of this work we adopt the following notational conventions :

$\forall$ c $\in$ $\mathbb{CLK}$, n $\in$ NUM : c_n denotes extract(c,n) (5.5)

$\forall$ c $\in$ $\mathbb{CLK}$, n $\in$ NUM, ρ $\in$ **ENV**, E_n $\in$ Exp :
  $\langle t_n, v_n \rangle$ denotes $\mathcal{E}_V$ $[\![E_n]\!]$ ρ c_n

It will often prove useful to ensure that all the timestamps in a clock are greater than a specific time and this is done by the ageing function clockafter.

```
clockafter : CLK x NUM → CLK                              (5.6)
clockafter = λ(c,n) . from(c,n)
```

This completes the definitions of "utility" functions used in specifying the semantics of **Ruth**.

5.3 The Standard Subset Of Ruth

5.3.1 Constants and identifiers

The simplest **Ruth** expression is the constant integer, boolean or string expression, for example `10`, `true` or `'A string'`. Taking booleans as an example :

$$\mathcal{E}_V [\![\texttt{true}]\!]\ \rho\ c = \texttt{after(<0,true>, c)} \tag{5.7}$$
$$\mathcal{E}_V [\![\texttt{false}]\!]\ \rho\ c = \texttt{after(<0,false>, c)}$$

When a constant is evaluated the result is available almost immediately; operationally a single "load constant" instruction is usually the most that will be required. Thus the semantic timestamp of the result of a constant reference is taken to be the first time on the clock `c`. Rather than deal with the clock explicitly the `after` function is used; giving the value argument to `after` a semantic timestamp of `0` ensures that the first time in the clock will be returned. This technique for avoiding explicitly dealing with the clock in semantic equations will be used wherever possible in this work.

Identifier reference is defined as follows

$$\mathcal{E}_V [\![\texttt{I}]\!]\ \rho\ c = \texttt{after(lookup(I,}\rho\texttt{), c)} \tag{5.8}$$

Note the ageing of the result to allow for the time taken to look up the identifier in the environment; operationally to load the required value from memory.

5.3.2 Binary Operations

In Section 5.1 the semantics of the boolean `And` operator were given informally `(5.1)` as an example of the semantics of binary operators in **Ruth**. Using the apparatus introduced in Section 5.2 the definition can now be given in a more formal manner and this is done overleaf.

$$
\begin{array}{l}
\mathcal{E}_V [\![\mathtt{E}_1\ \mathbf{And}\ \mathtt{E}_2]\!]\ \rho\ \mathtt{c} \\
\quad = \mathtt{after}\ (\\
\qquad \ll \langle t_1, v_1 \rangle \\
\qquad\qquad \rightarrow \ll \langle t_2, v_2 \rangle \rightarrow \langle \mathtt{max}(t_1, t_2),\ v_1 \wedge v_2 \rangle \gg \\
\qquad \gg, \\
\qquad \mathtt{extract}(\mathtt{c}, 0)) \\
\qquad \mathbf{where} \\
\qquad \langle t_1, v_1 \rangle = \mathcal{E}_V [\![\mathtt{E}_1]\!]\ \rho\ \mathtt{extract}(\mathtt{c}, 1) \\
\qquad \langle t_2, v_2 \rangle = \mathcal{E}_V [\![\mathtt{E}_2]\!]\ \rho\ \mathtt{extract}(\mathtt{c}, 2)
\end{array}
\tag{5.9}
$$

or, written in terms of the conventions given in (5.5)

$$
\begin{array}{l}
\mathcal{E}_V [\![\mathtt{E}_1\ \mathbf{And}\ \mathtt{E}_2]\!]\ \rho\ \mathtt{c} \\
\quad = \mathtt{after}\ (\\
\qquad \ll \langle t_1, v_1 \rangle \\
\qquad\qquad \rightarrow \ll \langle t_2, v_2 \rangle \rightarrow \langle \mathtt{max}(t_1, t_2),\ v_1 \wedge v_2 \rangle \gg \\
\qquad \gg, \\
\qquad c_0)
\end{array}
\tag{5.10}
$$

Note that in this definition a different sub-clock is used to evaluate each of the arguments, $\mathtt{E}_1$ and $\mathtt{E}_2$. Thus the semantics place no restriction on the order in which the arguments to binary operations are evaluated, though they must obviously be evaluated before the result is produced.

Equation (5.10) allows sequential or parallel evaluation but, by using the `clockafter` function, it is simple to specify strictly sequential evaluation.

$$
\begin{array}{l}
\mathcal{E}_V [\![\mathtt{E}_1\ \mathbf{And}\ \mathtt{E}_2]\!]\ \rho\ \mathtt{c} \\
\quad = \mathtt{after}\ (\\
\qquad \ll \langle t_1, v_1 \rangle \\
\qquad\qquad \rightarrow \ll \langle t^2, v^2 \rangle \rightarrow \langle t^2,\ v_1 \wedge v^2 \rangle \gg \\
\qquad \gg, \\
\qquad c_0) \\
\qquad \mathbf{where} \\
\qquad \langle t_1, v_1 \rangle = \mathcal{E}_V [\![\mathtt{E}_1]\!]\ \rho\ c_1 \\
\qquad \langle t^2, v^2 \rangle = \mathcal{E}_V [\![\mathtt{E}_2]\!]\ \rho\ \mathtt{clockafter}(c_2, t_1)
\end{array}
\tag{5.11}
$$

Using `clockafter` to provide a clock which only contains time values bigger than the time of evaluation of $\mathtt{E}_1$ specifies that the evaluation of $\mathtt{E}_2$ must be performed with a semantic clock which only contains times later than t_1, the time at which evaluation of E_1 was completed. This does not force an implementation to evaluate E_1 and E_2 sequentially but operationally it would appear that the simplest way to ensure that evaluation of E_2 is performed with times strictly greater than t_1 would be to evaluate them sequentially.

Although indicating desired evaluation orders in this way is simple, evaluation order is outside the scope of the descriptive semantics being constructed here. Consequently equation (5.10) will be taken to be that defining the semantics of boolean **And**.

Note that although only the **And** primitive has been considered here all the other binary operations in **Ruth** (i.e. boolean **Or**, equality and the arithmetic operators) are defined in the same way as **And** with the relevant operator being substituted for $\wedge$ in (5.10). Their definitions can be found in Appendix 2.

5.3.3 **If...Then...Else...**

$$\begin{array}{l} \mathcal{E}_V [\![\;\textbf{If}\; E_1 \;\textbf{Then}\; E_2 \;\textbf{Else}\; E_3 \;]\!]\; \rho\; c \\ \quad = \ll \langle t_1,v_1\rangle \\ \qquad\qquad \to (v_1 \to \mathcal{E}_V [\![E_2]\!]\; \rho\; \mathrm{clockafter}(c_2,t_1), \\ \qquad\qquad\qquad\qquad \mathcal{E}_V [\![E_3]\!]\; \rho\; \mathrm{clockafter}(c_3,t_1) \\ \qquad\qquad\quad) \\ \quad \gg \\ \quad \textbf{where} \\ \quad \langle t_1,v_1\rangle = \mathrm{after}(\mathcal{E}_V [\![\; E_1 \;]\!]\; \rho\; c_1,\; c_0) \end{array} \tag{5.12}$$

Note the use of clockafter to specify that evaluation of the **Then** and **Else** expressions should not be begun until after the boolean value defined by E_1 has been evaluated and a decision between **Then** and **Else** can be made. If clockafter were omitted an implementor of **Ruth** would have the freedom to evaluate E_1, E_2 and E_3 in parallel; since the result of either E_2 or E_3 will not be required some of this work would be discarded. When working with real-time systems it seems undesirable to allow processing resources to be consumed by work which may be unnecessary since this may result in deadlines not being met; this is prevented by the use of `clockafter`.

5.3.4 Function definition and application

When **FUNC**, the domain of functions, was defined in the last section, we saw that **Ruth** functions take a semantic clock as an implicit argument; this clock is supplied when the function is applied and is used to facilitate clock-driven timing.

$$
\begin{aligned}
&\mathcal{E}_V [\![\; \mathbf{lambda} \; (I_0 ... I_n) . \; E \;]\!] \; \rho \; c \qquad\qquad\qquad\qquad (5.13)\\
&\quad = \mathrm{after}(\langle 0, f \rangle, c)\\
&\qquad \mathbf{where}\\
&\qquad f = \lambda\delta_c . \lambda\delta_0 . \; ... \; \lambda\delta_n . \mathcal{E}_V [\![E]\!] \; (\rho \; \& \; [I_0 \rightarrow \delta_0] \; \& \; ... \; \& \; [I_n \rightarrow \delta_n]) \; \delta_c
\end{aligned}
$$

Here δ_c is the place holder for the semantic clock argument and the expression E will be evaluated using the clock to be supplied as its semantic clock upon function application. Note that the result of the **lambda** expression is assumed to be available at the first time on the clock c, or in other words at some time after its evaluation is begun.

$$
\begin{aligned}
&\mathcal{E}_V [\![\; E_1 \; (E_2 ... E_n) \;]\!] \; \rho \; c \qquad\qquad\qquad\qquad (5.14)\\
&\quad = \langle\!\langle \; \langle t_1, f_1 \rangle\\
&\qquad\quad \rightarrow f_1 \; \mathrm{clockafter}(c_0, t_1) \; \langle t_2, v_2 \rangle \; ... \; \langle t_n, v_n \rangle\\
&\qquad \rangle\!\rangle\\
&\qquad \mathbf{where}\\
&\qquad \langle t_1, f_1 \rangle = \mathrm{after}(\mathcal{E}_V [\![\; E_1 \;]\!] \; \rho \; c_1, \; c_0)
\end{aligned}
$$

When a function is applied the semantic clock supplied to it is constrained by the use of `clockafter` to contain times strictly greater than the time at which the function is evaluated; operationally, evaluation of the function body cannot take place until the code for the function body is loaded from memory. Note that each of the "real" (i.e. explicitly user defined) arguments is evaluated with a unique clock; thus function arguments may be evaluated in parallel or not, as an implementor wishes. Also note that the semantic timestamps of the "real" arguments have no bearing upon when evaluation of the function body starts. Operationally this is in line with lazy evaluation : the time of availability (semantic timestamp) of the result of a function application will only depend on the times of availability of those arguments required to produce that result. If an argument is referenced its semantic timestamp will be taken into account in the semantic timestamps of the result of any expression referring to it. If the identifier is not referenced then its semantic timestamp is irrelevant to the result of the function application.

5.3.5 Identifier definitions

Identifiers in **Ruth** are defined by being bound to the value of an expression by a **where** or **whererec** construct.

$$\begin{aligned} &\mathcal{E}_V [\![\ \mathtt{E}\ \mathbf{where}\ \mathtt{D}\ \mathbf{endwhere}\ ]\!]\ \rho\ c \\ &\quad = \mathcal{E}_V[\![\mathtt{E}]\!]\ (\mathcal{E}_D[\![\mathtt{D}]\!]\ \rho\ c_1)\ c_0 \end{aligned} \qquad (5.15)$$

$$\begin{aligned} &\mathcal{E}_V [\![\ \mathtt{E}\ \mathbf{whererec}\ \mathtt{D}\ \mathbf{endwhererec}\ ]\!]\ \rho\ c \\ &\quad = \mathcal{E}_V[\![\mathtt{E}]\!]\ \rho'\ c_0 \\ &\qquad \mathbf{where} \\ &\qquad \rho' = \mathtt{fix}(\lambda\rho''.\rho\ \&\ \mathcal{E}_D[\![\mathtt{D}]\!]\ \rho''\ c_1) \end{aligned}$$

Here `fix` is the usual fixed point function and is defined in Appendix 1, `D` is a member of `Dec`, the syntactic domain of declarations, and $\mathcal{E}_D$ is the meaning function for declarations which is defined

$$\mathcal{E}_D\ :\ \mathtt{Dec} \rightarrow \mathbf{ENV} \rightarrow \mathbb{CLK} \rightarrow \mathbf{ENV} \qquad (5.16)$$

$$\begin{aligned} &\mathcal{E}_D\ [\![\ \mathtt{I} = \mathtt{E}\ ;\ \mathtt{D}\ ]\!]\ \rho\ c = [\mathtt{I} \rightarrow \mathcal{E}_V[\![\mathtt{E}]\!]\ \rho\ c_0]\ \&\ \mathcal{E}_D[\![\mathtt{D}]\!]\ \rho\ c_1 \\ &\mathcal{E}_D\ [\![\ ]\!]\ \rho\ c = \emptyset \end{aligned}$$

Note that each declaration in a list of declarations is evaluated using a unique clock, thus allowing for them to be evaluated in parallel if desired.

5.3.6 S-expression manipulation

Atomic and `Nil` valued s-expressions have a straightforward semantics and the reader is referred to Appendix 2 for details. Our interest here is focused upon s-expressions which are **Cons**ed pairs.

$$\mathcal{E}_V [\![\ \textbf{Cons}\ (E_1,\ E_2)\ ]\!]\ \rho\ c \qquad\qquad (5.17)$$
$$= \mathtt{after}(<0, \mathtt{lift}([<t_1,v_1>,<t_2,v_2>])>, c_0)$$

The time at which E_1 and E_2 are evaluated has no bearing upon when the s-expression is evaluated because of lazy evaluation and thus the first value from the clock provides the semantic timestamp for the pair. Note that because the head and tail of the pair may be independently evaluated they retain their own semantic timestamps in the **Cons**ed pair. When **Head** or **Tail** is applied to the pair these semantic timestamps are used to define when the result is becomes available.

$$\mathcal{E}_V [\![\ \textbf{Head}\ (E_1)\ ]\!]\ \rho\ c \qquad\qquad (5.18)$$
$$\begin{array}{l}
= \mathtt{after}\ (\\
\quad \langle\!\langle\ <t_1,v_1> \\
\qquad \rightarrow\ (\mathtt{Cases}\ v_1\ \mathtt{of} \\
\qquad\qquad \mathtt{lift}([<t_H,s_H>,<t_T,s_T>]) \rightarrow <\max(t_1,t_H),s_H> \\
\qquad\qquad \mathtt{else} \qquad\qquad\qquad\qquad\quad\ \rightarrow <\infty,\bot> \\
\qquad\quad) \\
\quad \rangle\!\rangle, \\
\quad c_0)
\end{array}$$

The result of the **Head** becomes available after the maximum of t_1 and t_H; operationally the result of a **Head** cannot be available until both the **Cons** and the head element itself have been evaluated. Note that if the argument to **Head** is not a pair the non-terminating computation, $<\infty,\bot>$, results. **Tail** is similar and its definition is given overleaf.

```
$\mathcal{E}_V$ ⟦ Tail ($E_1$) ⟧ ρ c                                    (5.19)
   = after (
     « <$t_1$,$v_1$>
         → (Cases $v_1$ of
                lift([<$t_H$,$s_H$>,<$t_T$,$s_T$>]) → <max($t_1$,$t_T$),$s_T$>
                else                          → <∞,⊥>
            )
     »,
     $c_0$)
```

This completes the definitions of the standard subset of **Ruth**. The next section gives the specification of the semantics of those parts of the language directly concerned with the programming of real-time systems.

5.4 The Real-Time Subset Of Ruth

5.4.1 Channel construction and reference

When dealing with channels in the semantics of **Ruth** we will normally be dealing with elements of $\mathbb{CHAN}$ embedded in $\mathbb{VAL}$, that is, with objects of the form $<t_v,<t_{ch},[a,\mathrm{lift}(ch)]>>$ and $<t_v,<t_{ch},\perp>>$. Here t_v is the semantic timestamp and, in the first case, t_{ch} is the message timestamp of the first message a, and in the second a denotation of a time by which the channel has not yet produced a message.

As was commented in Chapter 4 there need be no connection between the values of t_v and t_{ch}. For example, a process references a channel from another process, or from the external environment, via the corresponding identifier given in its **Input** list. In this case the value of t_v denotes when the receiving process has identified which channel it is attempting to access. Obviously this is totally independent of t_{ch}, the time at which the sending process puts the first message in the channel. A simpler case is that a channel may be be created much earlier than the timestamp on its first message. Finally, the ageing of the results of, for example, references to identifiers bound to channels in environments, will result in the value of t_v growing larger without changing that of t_{ch}.

However there is one point at which it is desirable to enforce a relationship between t_v and t_{ch} and that is when channels are created via a **ConsCh**. The required relationship is that t_v be no bigger than t_{ch}, or in other words that the channel has not been created after the time denoted by its first message timestamp. The desired interpretation of message timestamps in channels is that they denote the time at which the message containing them is first available to be referenced. Clearly any channel which is created at time t_v cannot be referenced before t_v and therefore neither can any of its messages. It is clearly nonsense to allow a program to decide at time 10 (t_v) that it will produce a message at time 5 (t_{ch}).

Thus to allow a channel to be created of the form

$$<t_v,\ <t_{ch},[a,ch]>\ >\ \textbf{where}\ t_v > t_{ch} \qquad (5.20)$$

would obviously be undesirable since the message cannot be used until after t_v (and thus after t_{ch}). Of course after a channel has been constructed no such restriction can be, or should be, enforced for the reasons given above. After a channel has been constructed the desired restriction is simply that the channel message cannot be referenced before time t_{ch}; when the message actually is referenced is the responsibility of whatever is receiving the channel. The implicit assumption is that external hardware devices will always use messages as soon as they receive them.

Operationally a **ConsCh** can be viewed as constructing its result in two parts : firstly, the atomic data value and the numerical timestamp arguments are evaluated to form the initial message; after the message is sent the third argument is evaluated to provide the rest of the channel. The definition of **ConsCh** given below is also partitioned in this way : (5.21) below defines how the first message of the channel is evaluated; the rest of the channel, denoted by rest in (5.21) is dealt with in (5.22).

$$
\begin{array}{l}
\mathcal{E}_V\ [\![\ \textbf{ConsCh}\ (E_1,\ E_2,\ E_3)\ ]\!]\ \rho\ c \qquad\qquad\qquad\qquad\qquad\qquad (5.21)\\
\quad = \mathrm{after}\ (\\
\quad\quad \langle\!\langle\ \langle t_1, v_1\rangle\\
\quad\quad\quad\quad \rightarrow\ \langle\!\langle\ \langle t_2, v_2\rangle\ \rightarrow\ (v_2\ \le\ t\ \rightarrow\ \langle\infty,\bot\rangle,\ \langle t, ch\rangle)\ \rangle\!\rangle\\
\quad\quad \rangle\!\rangle,\\
\quad\quad c_0)\\
\quad\quad \textbf{where}\\
\quad\quad \langle t, ch\rangle\ =\ \mathrm{after}(\langle \max(t_1, t_2),\ \langle v_2, [v_1, \mathrm{lift}(\mathrm{rest})]\rangle\ \rangle, c_0)
\end{array}
$$

Note that unless the timestamp value, v_2, denotes a time at or after completion of the **ConsCh** the $\langle\infty,\bot\rangle$ element of $\mathbb{VAL}$ results.

$$
\begin{array}{l}
\mathrm{rest}\ =\ \langle t'' - (v_2+1),\ v''\rangle \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (5.22)\\
\quad\quad \textbf{where}\\
\quad\quad \langle t'', v''\rangle\ =\ \langle\!\langle\ \langle t_3, v_3\rangle\\
\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow\ \langle\!\langle\ v_3\ :\ \langle t', [a', \mathrm{lift}(ch')]\rangle\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow\ \ (t'\ge t_3\ \wedge\ t'\ >\ v_2\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow\ \langle t', [a', \mathrm{lift}(ch')]\rangle,\ \langle\infty,\bot\rangle\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \rangle\!\rangle\\
\quad\quad\quad\quad\quad\quad \rangle\!\rangle
\end{array}
$$

Note that E_3 evaluates to an element of $\mathbb{CHAN}$ embedded in $\mathbb{VAL}$ and not merely an element of $\mathbb{CHAN}$. If the timestamp of the first message in v_3, t', is no bigger than v_2 or is less than t_3 then the value of `rest` is the $<\infty,\bot>$ element of $\mathbb{CHAN}$. t' must be bigger than v_2 to maintain the restriction that message timestamps in a channel must be strictly increasing. t_3 is the time at which the channel denoted by E_3 was referenced by the **ConsCh**, which will be after the time at which the channel was first created because of ageing. If t' is smaller than t_3 then at least the first message in the channel will not be available for use at the time denoted by its timestamp. Finally, note that the value of the first message timestamp in `rest` is adjusted in line with the incremental interpretation of message timestamps.

Channels are referenced via **HeadCh**, **Time** and **TailCh**.

$$
\begin{array}{l}
\mathcal{E}_V [\![\ \textbf{HeadCh}\ (E_1)\ ]\!]\ \rho\ c \qquad\qquad\qquad\qquad (5.23) \\
\quad = \texttt{after}\ (\\
\qquad \langle\!\langle\ <t_1,v_1> \\
\qquad\qquad \rightarrow \langle\!\langle\ v_1\ :\ <t,[a,\texttt{lift}(ch)]> \rightarrow <\max(t_1,t),a>\ \rangle\!\rangle \\
\qquad \rangle\!\rangle, \\
\qquad c_0)
\end{array}
$$

The result of a **HeadCh** becomes available after the maximum of the semantic timestamp and the message timestamp of the first message in the channel. Operationally the result of a **HeadCh** cannot become available until both the channel has been computed and it has produced a message. The definition of **Time** is similar.

$$
\begin{array}{l}
\mathcal{E}_V [\![\ \textbf{Time}\ (E_1)\ ]\!]\ \rho\ c \qquad\qquad\qquad\qquad (5.24) \\
\quad = \texttt{after}\ (\\
\qquad \langle\!\langle\ <t_1,v_1> \\
\qquad\qquad \rightarrow \langle\!\langle\ v_1\ :\ <t,[a,\texttt{lift}(ch)]> \rightarrow <\max(t_1,t),t>\ \rangle\!\rangle \\
\qquad \rangle\!\rangle, \\
\qquad c_0)
\end{array}
$$

TailCh is slightly more complex since the first message timestamp in the tail of the channel must be replaced with the time it actually represents under the incremental interpretation of message timestamps.

$$
\begin{array}{l}
\mathcal{E}_V [\![\ \mathbf{TailCh}\ (E_1) \ ]\!] \ \rho \ c \hfill (5.25)\\
\quad = \texttt{after} \ (\\
\qquad \langle\!\langle \ <t_1, v_1> \\
\qquad\qquad \rightarrow \ \langle\!\langle \ v_1 \ : \ <t, [a, \texttt{lift}(ch)]> \\
\qquad\qquad\qquad\qquad \rightarrow \ <\max(t_1, t), \ <t'+t+1, v'> \ > \\
\qquad\qquad\qquad\qquad\quad \mathbf{where} \\
\qquad\qquad\qquad\qquad\quad <t', v'> = \langle\!\langle \ ch \ : \ <t_r, [a_r, \texttt{lift}(ch_r)]> \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow \ <t_r, [a_r, \texttt{lift}(ch_r)]> \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \rangle\!\rangle \\
\qquad\qquad\quad \rangle\!\rangle \\
\qquad \rangle\!\rangle, \\
\qquad c_0)
\end{array}
$$

The time at which the result of a **TailCh** becomes available is `after` the maximum of the semantic timestamp of the channel and the first message timestamp in the channel. Operationally, the first message in a channel cannot be discarded until both the channel has been identified and its first message received.

5.4.2 The **Ready** test

One reason for using herring-bone domains was to enable the **Ready** function to be specified as a non-blocking test on a channel. The basic principle is that if a channel is undefined at a time greater than the **Ready** test is checking for then **Ready** can timeout the channel. It must be remembered that the **Ready** test is a test upon user defined message timestamps and not upon the semantic timestamps. If a **Ready** test is supplied with a $<t, \perp>$ element of $\mathbb{VAL}$ it cannot timeout on the basis of `t` since `t` has no connection with the values of the message timestamps. Operationally a $<t, \perp>$ element of $\mathbb{VAL}$ corresponds to the situation that the expression identifying which channel is to be tested has not yet been evaluated. Clearly a **Ready** test cannot timeout the channel if it has not yet even identified which channel it is testing.

The definition of **Ready** is given overleaf and can most easily be explained by considering each labelled case of the `Match ... with ...` construct in turn.

```
      $\mathcal{E}_V$ ⟦ Ready ($E_1$, $E_2$) ⟧ ρ c                          (5.26)
        = after (
            ≪ <$t_1$,$v_1$>
                  → ≪ <$t_2$,$v_2$>
                          → (Match $v_1$ with
(i)                             <t,⊥>
                                  → (t > $v_2$
(ia)                                    →<max($t_1$,$t_2$,$v_2$),false>,
(ib)                                      <max($t_1$,$t_2$,t),⊥>
                                    )
(ii)                            <t,[a,lift(ch)]>
                                  → (t > $v_2$
(iia)                                   →<max($t_1$,$t_2$,$v_2$),false>,
(iib)                                     <max($t_1$,$t_2$,t),true>
                                    )
                             )
                  ≫
            ≫,
            $c_0$)
```

(i) The channel is <t,⊥> : no message has been produced up to time t, though a message could still be produced at time t. There are two sub-cases to consider.

(ia) t > v_2 : the channel did not produce a message before, or at, the time being tested for and consequently the channel can be timed out and the result of the **Ready** test is false. The result is produced at some time after the maximum of the time at which the two arguments to the **Ready** test are calculated and the time which is being tested for since, operationally, **Ready** cannot return a result until its arguments have been calculated and the timeout time has passed.

(ib) t ≤ v_2 : the channel has not yet produced a message but it cannot be timed out since the testing time has not yet passed. The result of the **Ready** test is thus undefined with a semantic timestamp after the maximum of the time at which the two arguments to the **Ready** test are calculated and the time by which the channel has not produced a message.

(ii) The channel is <t, [a,lift(ch)]> : a message has been produced at time t.

Once again there are two sub-cases to consider.

(iia) $t > v_2$: the channel did not produce the message before, or at, the time being tested for. The result is identical to that for case (ia) since the channel will be timed out by the **Ready** test.

(iib) $t \leq v_2$: the channel produced the message before, or at, the time being tested for. The result of the **Ready** test is true at the maximum of the time at which the two arguments to the **Ready** test are calculated and the time at which the message arrived since, operationally, **Ready** will return a result as soon as its arguments have been calculated and the message arrives.

(ia) and (iia) are the most interesting since they show how the non-blocking, timeout behaviour of **Ruth** is specified in the semantics.

5.4.3 Tuple construction and reference

The only other channel operators are those connected with tuples. Tuple construction is defined as follows.

$$\mathcal{E}_V [\![\{E_1 \ldots E_n\}]\!] \rho\ c \qquad (5.27)$$
$$= \mathrm{after}(\langle 0, \{\langle t_1, ch_1\rangle, \ldots, \langle t_n, ch_n\rangle\}\rangle, c_0)$$

The times at which the channels are identified have no bearing on when the tuple is actually constructed. The result of a tuple construction becomes available at the first time read from the clock; operationally at some time after evaluation of the tuple construction commences.

Elements of a tuple are referenced via the ! operator which is defined overleaf.

$$
\begin{array}{ll}
\mathcal{E}_V [\![\mathtt{E}_1 \; ! \; \mathtt{E}_2]\!] \; \rho \; c & (5.28) \\
\quad = \mathtt{after} \; (& \\
\qquad \langle\!\langle \; <t_1, v_1> & \\
\qquad\qquad \rightarrow \; \langle\!\langle \; <t_2, v_2> \; : \; <t_2, i> & \\
\qquad\qquad\qquad \rightarrow \; (1 \leq i \leq m \rightarrow <\max(t_1, t_2, t^i), ch^i>, <\infty, \perp>) & \\
\qquad\qquad\qquad \mathbf{where} & \\
\qquad\qquad\qquad \{<t^1, ch^1>, \; \ldots, \; <t^m, ch^m>\} = v_1 & \\
\qquad\qquad \rangle\!\rangle & \\
\qquad \rangle\!\rangle, & \\
\qquad c_0) &
\end{array}
$$

The result of a tuple reference becomes available at some time after the tuple is constructed and the channel being referenced is identified. Note that tuple elements are indexed from one and if the index number defined by E_2 is not within the required range the result is $<\infty, \perp>$.

5.4.4 Process definition and application

The major reason for considering process definition and application here is to show how the semantics of **Ruth** correctly specifies the essential real-time requirement of the language : that message timestamps denote the earliest time at which messages become available for use at their destinations. Such a check was used in the definition of **ConsCh** and, since a new message in a channel is available for use within the process performing the **ConsCh** as soon as the **ConsCh** is completed, that check is sufficient to ensure the real-time requirement provided the channel is never communicated to another process. If the channel being constructed is being communicated to another process then simply checking when the channel is constructed will not be sufficient.

To illustrate this consider the following process definition.

```
Process P Input I Clock c Is {I} ;                    (5.29)
```

When I is constructed in its origin process its message timestamps will be checked to ensure that its messages can be delivered to P within the deadlines they denote. P simply passes I on unchanged so that there is a possibility these deadlines will have passed before I's messages can be delivered to their ultimate destination. A check must be made in P to ensure this does not occur and the obvious place to make this check is when a message is transmitted from P to another process. In the semantics this is modelled in the definition of process application via the function filter (see (5.32) below).

Firstly, let us consider process definition which is defined using the semantic function $\mathcal{E}_P$ (evaluate process).

$$\mathcal{E}_P : \texttt{Proc} \rightarrow \textbf{ENV} \qquad (5.30)$$

$$\mathcal{E}_P [\![\textbf{Process}\ \texttt{I}\ \textbf{Input}\ \texttt{I}_1 \ldots \texttt{I}_n\ \textbf{Clock}\ \texttt{I}_c\ \textbf{Is}\ \texttt{E}\ ]\!]$$
$$= [\texttt{I} \rightarrow \langle 0, \texttt{f} \rangle]$$
$$\textbf{where}$$
$$\texttt{f} = \lambda\delta_{sc}.\lambda\delta_1, \ \ldots\ \lambda\delta_n.\lambda\delta_{pc}.$$
$$\mathcal{E}_V[\![\texttt{E}]\!]\ ([\texttt{I}_1 \rightarrow \delta_1]\ \&\ldots\&\ [\texttt{I}_n \rightarrow \delta_n]\ \&\ [\texttt{I}_c \rightarrow \delta_{pc}])\ \delta_{sc}$$

Here Proc is the syntactic domain of process definitions. Evaluation of a process definition results in an environment in which the process function it denotes is bound to the process's name. Because we wish to store the process function in an environment we must turn it into an element of $\mathbb{VAL}$ by adding a semantic timestamp of 0 indicating that the process function is defined at the start of evaluation of the program. Operationally a process function is loaded before the program begins to run and therefore is available for access at time 0. δ_{sc} and δ_{pc} denote that two clocks are required as arguments to the process function : δ_{sc} is the semantic clock which is used to provide timing information to the semantics of the process function; δ_{pc} is the program clock which supplies timing information to the **Ruth** process itself. These two clocks will be provided when the process function is applied.

Process application is defined overleaf using the semantic function $\mathcal{E}_{PA}$ (evaluate process application).

```
$\mathcal{E}_{PA}$ : Pr-App → ENV → $\mathbb{CLK}$ → ENV                         (5.31)

$\mathcal{E}_{PA}$ ⟦ $I^1 \ldots I^m$ = I ($I_1 \ldots I_n$) ⟧ ρ c
    = [$I^1$ → <0,$ch^1$>] & ...& [$I^m$ → <0,$ch^m$>]
      where
      $ch^j$ = « <$t_j$,$ch_j$> → filter($ch_j$,0,clockafter($c_j$,max(t',$t_j$))) »
      <t',{<$t_1$,$ch_1$>, ..., <$t_m$,$ch_m$>}>
         = « lookup(I,ρ) : <t,f>
                → f sc $i_1 \ldots i_n$ <0,pc>
                  where
                  $i_k$ = lookup($I_k$,ρ)
                  sc = extract($c_0$,0)
                  pc = extract($c_0$,1)
           »
```

Here Pr-App is the syntactic domain of process applications. The process function is retrieved from the environment produced by (5.30) and applied to its input channels. The process application also extracts two sub-clocks from c_0 (converting one of them into an element of $\mathbb{VAL}$, for use as the program clock, by adding a semantic timestamp of 0) for the process function to use as its semantic and program clocks. Each of the output channels is stored in environment which results from the process application. This environment is referenced by the definition of process configurations, given in Appendix 2, to produce the result of a **Ruth** program. In order to store the channels in an environment they must be embedded in $\mathbb{VAL}$ and this requires the addition of an extra semantic timestamp. The value of this extra semantic timestamp is 0 since the channels can be identified by their destination processes as soon as the program starts running.

The earliest the process will be able to output any message is t', the time at which the tuple of output channels is constructed. Consequently, when the real-time constraint on each output channel's messages is checked by the function filter the starting time for the checking is max(t',t_j); t_j is the time at which the process identifies the output channel j. filter is defined overleaf.

```
filter : ℂℍ𝔸ℕ x NUM x ℂ𝕃𝕂 → ℂℍ𝔸ℕ (5.32)
filter =  λ(ch,n,c).<t-n,v>
          where
          <t,v> = « c : <$t_c$,[$l_c$,$r_c$]>
                      → « ch : <t,[a,lift(rest)]>
                              → ($t_c$ ≤ t + n
                                   →<t + n,[a,lift(rest')]>, <∞,⊥>
                                    where
                                    rest' = filter(rest,t+n+1,
                                                   from($l_c$,$t_c$))
                                )
                        »
                  »
```

Each message timestamp is checked to ensure that it denotes a time greater than the current time by testing it against the first time in the clock. Note the use of the argument n to handle the incremental nature of channel timestamps. n denotes the value that must be added to the first message timestamp in the channel to provide the actual time that it denotes; thus n is initially 0. If the actual time denoted by the first message timestamp in the channel is no less than the current time, t_c, then the first message in the channel may be output; otherwise the <∞,⊥> channel results.

5.4.5 Program clocks

As with the channel domain ℂℍ𝔸ℕ, the clock domain ℂ𝕃𝕂 is embedded in the domain of expressible values 𝕍𝔸𝕃 and thus a clock has two types of timestamp, the semantic timestamp denoting when the clock is identified and the "clock" timestamps denoting the time values in the clock. There need be no connection between the semantic timestamp associated with a clock and the first clock timestamp since the clock may already have been instantiated by previous references to it.

Ruth has two operators on program clocks : `HeadClk` and `TailClk`.

$$\begin{array}{l}
\mathcal{E}_V [\![\textbf{HeadClk} \; (E_1)]\!] \; \rho \; c \qquad\qquad\qquad\qquad (5.33) \\
\quad = \mathtt{after(} \\
\quad\quad \ll \langle t_1, v_1 \rangle \\
\quad\quad\quad\quad \rightarrow \ll v_1 \; : \; \langle t, [l, r] \rangle \rightarrow \langle \max(t_1, t), t \rangle \gg \\
\quad\quad \gg, \\
\quad\quad c_0)
\end{array}$$

The time of availability of a clock reference must be after the maximum of the time of availability of the clock and the actual time value read from it. This models the fact that a value read from a clock is out of date as soon as it is obtained. A real-time programmer should never fall into the trap of assuming that a value read from a clock denotes the current value of real-time; it denotes what the real-time was at some point in the (more or less) recent past when the clock was actually referenced. In **Ruth** this situation can be made worse by references to clocks which have already had their values instantiated, and so contain values for times from the past. The conclusion must be that real-time programmers in general, and **Ruth** programmers in particular, should treat real-time clock values with extreme caution.

$$\begin{array}{l}
\mathcal{E}_V [\![\textbf{TailClk} \; (E_1)]\!] \; \rho \; c \qquad\qquad\qquad\qquad (5.34) \\
\quad = \mathtt{after(} \\
\quad\quad \ll \langle t_1, v_1 \rangle \\
\quad\quad\quad\quad \rightarrow \ll v_1 \; : \; \langle t_c, [l_c, r_c] \rangle \rightarrow \langle \max(t_c, t_1), \; \mathrm{from}(l_c, t_c) \rangle \gg \\
\quad\quad \gg, \\
\quad\quad c_0)
\end{array}$$

Note here that the "tail" of the clock has a time of availability `after` the maximum of the identification time of the whole clock and the first clock timestamp. Note also that the value in the root of the left sub-tree must be adjusted to be the time it actually denotes under the incremental interpretation of clock values before this sub-tree is returned as the result.

5.5 Conclusion

The purpose of this chapter has been to give a definition of the semantics of **Ruth**. Such a semantics requires the use of a model that allows timing information to be expressed and, following [Broy 83], we chose to use a denotational semantic model based upon herring-bone domains. An important consideration in specifying the semantics of a real-time language is how to model the durations of machine operations. The method chosen here was that of clock-driven timing because it offers a way of modelling operation durations and data dependencies without the necessity of specifying low-level operational detail.

Without herring-bone domains it would have been impossible to define the non-blocking nature of the `Ready` test; the best that could have been done would have been to use implicit time determinance and natural language. The same is true of the real-time restrictions upon channel construction and channel messages crossing process boundaries.

In the next chapter we turn from the theoretical issues which have occupied Chapters 4 and 5 towards the more practical concern of using **Ruth** to program a real-time system, in this case a real-time computer game.

Chapter 6 : **Using Ruth : A Real-Time Computer Game**

6.1 Introduction

Real-time computer games are a type of real-time system which are often ignored by researchers in the area. This is perhaps surprising since they are very useful vehicles for testing out different approaches to the problem of constructing real-time software. In a real-time computer game we are faced with the usual problem in real-time systems of making events happen at the right time. Although the deadlines within which a game must react are relatively large, because games interact with humans and not machinery, the consequences of failure to meet deadlines could be disastrous : noticeable delays will dissuade people from playing the game.

Whilst computer games impose the same type of real-time constraints as other real-time systems, they do have the advantage of being simpler to construct and test than others. For example an engine controller program would require the software designer to have information about the workings of the engine being controlled. A computer game controls only very simple hardware (i.e. input keys, a screen and a loud speaker) and, in a lot of cases, a standard VDU meets all the hardware requirements.

The game we shall consider in this chapter is called **Minesweep** and the complete text of the program is given in Appendix 4. The major area of interest is in seeing how **Ruth** copes with the real-time requirements of the game and consequently this implementation abstracts away from issues such as driving the screen display in favour of concentration on the real-time issues.

In the next section we shall describe the game and in Sections 6.3 through 6.6 the **Ruth** program which implements it. Finally, we relate what we have learned about **Ruth** from the exercise.

6.2 Minesweep

Minesweep is played on a fifteen by fifteen square board. We refer to the positions on the board as cells. There are three states which a cell may be in : null (indicated on the board by '-'); mine ('*'); and scoring ('1' to '9'). A typical board is shown below.

```
- - - - - - - - - 4 - - - - -          (6.1)
* - - - - - - - - - - - - - -
- - - - - - 3 - - 8 - - - * -
- - - - - - - - - - - - - - -
- - 6 - - - - * - - - - - - 2
- - - - - - - - 8 - - - - - -
- - - - - - - - - - - - - - -
1 3 - - - - - - 9 @ * - - - -
- - - - * - - - - - - - - - -
- - - - - - - 7 - 3 - - - - -
- - - 9 - - - - - - - - - - -
- - - - - - - - - - - - - - 5
- - - - - 6 * - - - - * - - -
- - - - - - - - 2 - - - - - -
* - - - - - - - - - - - 3 - -
```

Initially all the cells on the board are null. The player starts the game on the top,left hand corner cell and then moves around the board either horizontally or vertically, scoring points by landing on the numbered cells. If the player, indicated by '@' in the diagram above, moves one cell to his left he will score nine points. However if he moves one cell to his right the game will be over since that cell is a mine and will "blow up" any player who lands on it. If the player moves one cell up or down nothing happens since these cells are in the null state.

To make the game a little more difficult the cells periodically change state. Initially the interval between changes of cell state (that is, between the occurrence of one state change and the next) is (about) 20 seconds; this value is referred to as the **period**. In order to prevent all the cells changing simultaneously a random factor of ± half the period is added. Thus, initially, a cell will change state at random intervals of 20 ± 10 seconds (i.e. between 10 and 30 seconds). Each time a cell changes state it decreases its period by 0.25 seconds down to a limit of 2.5 seconds. Thus the longer the game continues the faster the cells change state until they reach the limiting period of 2.5 ± 1.25 seconds between state changes.

If the player is on a cell which changes state he is unaffected. That is, if the cell becomes a scoring one he does not score any points and if it becomes a mine he is not blown up. The player must move onto a cell for its current state to have any effect on the progress of the game.

A cell may change into any one of the three states at random; its current state has no bearing on what it's next state will be. (Note that a cell's new state may be the same as it's previous state). There is always a 50% chance that a cell will be a scoring one but the chance of a cell being a mine (referred to as the **danger**) varies as the game progresses. Initially there is only a 5% danger but this increases by 0.5% each time the cell changes state until it reaches a limit of 40%. Consequently the chances of a cell being null are initially 45% but this declines to a limit of 10% as the game proceeds.

It only remains to specify the implementation's real-time performance requirements. A player may make moves as rapidly as he wishes, however the implementation need only respond to player moves at 0.2 second intervals; any extra moves will be queued. Thus the fastest the player can move on the screen is five times a second. Assuming that the player does not make more than five moves a second the response time to moves (the time between his making a move and that move appearing on the screen) should be 0.005 seconds. If the response time is large enough for the player to notice it this will detract from his enjoyment of the game; 0.005 seconds should be a small enough delay.

Of course, in a real-time system it is impossible to guarantee that specified deadlines will be met and recovery action in the case of failure must also be specified. In this case a failure to meet the 0.005 second window should result in the move appearing on the screen as soon as possible after that time. Although failure to meet the deadline is undesirable not displaying the move at all would be worse : a system which responds erratically to player inputs is bad enough, one which discards player moves whenever convenient would be disastrous.

Finally there is the obvious real-time constraint that cell state changes should appear on the screen at the time specified by the cell. However, in cases where this deadline is not met the state change will be ignored. Since a cell's new state may be the same as its old state anyway this is an acceptable situation. Rather than display a state change later than the time specified, the system assumes that no change of state occurred.

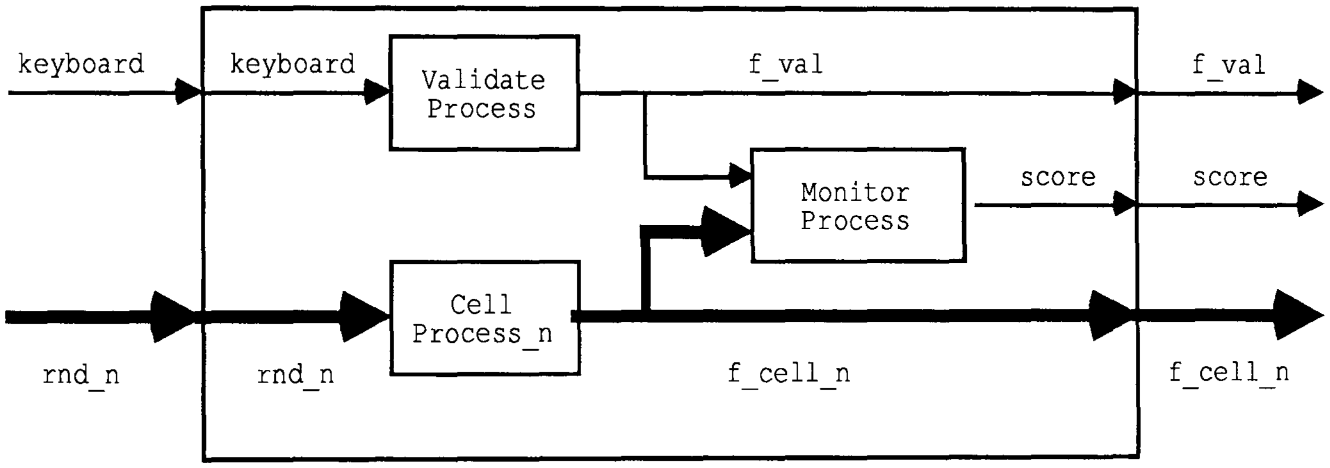
6.3 Overview of the implementation

6.3.1 System Configuration

The obvious way to implement the **Minesweep** board is as a set of processes, one for each cell. The first thing we note is that there are 225 cells on the **Minesweep** board. To avoid having to list all 225 cell processes and their associated channels we shall restrict this implementation to only 4 cells; this restriction does not effect the complexity of the system, merely the amount of processes and channels that must be listed in the configuration.

The overall structure of the system is as below.

Minesweep



In the above diagram the boxes represent processes and the arrows channels. Instead of explicitly including all four cell processes and their input and output channels a single `Cell_Process_n` box (for 1 ≤ n ≤ 4) is shown; in the same way the fat arrows represent the inputs and output channels, `rnd_n` and `f_cell_n` for each cell process.

This system configuration is thus

```
Configuration Minesweep                                        (6.2)
Output score, f_cell_1, f_cell_2, f_cell_3, f_cell_4, f_val
Input  keyboard, rnd_1, rnd_2, rnd_3, rnd_4
Is
score
  = Monitor_Process (f_cell_1,f_cell_2,f_cell_3,f_cell_4,f_val);

f_cell_1 = Cell_Process (rnd_1) ;
f_cell_2 = Cell_Process (rnd_2) ;
f_cell_3 = Cell_Process (rnd_3) ;
f_cell_4 = Cell_Process (rnd_4) ;
f_val    = Validate_Process (keyboard) ;
end.
```

6.3.2. System inputs and outputs

The **Minesweep** configuration takes the player's moves as input, via channel `keyboard`, in the form of ASCII characters, and produces six channels : an `f_cell_n` channel for each cell carrying the cell's state changes; `score`, which carries integers representing the player's score so far; and `f_val` carrying integers representing the cell the player has just moved to. We shall assume the existence of a hardware "process" which will display the original state of the screen (all cells null, score 0 and the player on the top left cell) and will then display the output from `score` and `f_cell_n` on the screen in the required format at the times denoted by the message timestamps. This hardware process will also display the player character, '`@`', on the cell indicated by the messages in `f_val`; it is also assumed that the hardware process will never overwrite the player character with a cell state in the case where the cell changes state whilst the player is on it.

It is simple to implement any screen driver we might require in **Ruth** but this would unnecessarily add to the complexity of our system so we shall not do so here.

The tables below define the input and output formats. Table `(6.3)` defines the characters the **Minesweep** system will receive from channel keyboard and the player moves they denote; table (6.4) defines the integers sent on the f_cell_n channels and the cell states they denote.

ASCII Character	**Player Move**	(6.3)
u	One cell upwards	
d	One cell downwards	
l	One cell left	
r	One cell right	

Integer	**Cell State**	(6.4)
-1	Null	
-2	Mine	
1 to 9	Scoring cell	

The system also takes four channels rnd_n ($1 \leq n \leq 4$) containing random integers in the range 1 to 1000. These are used to generate the cell state changes and the random part of the cells' periods; the rnd channels are assumed to be always **Ready**, that is, always able to provide a message. Effectively the rnd channels produce messages in a demand driven manner.

Finally, the end of the game, (when the player lands on a mine) is signalled to the external world by the sending of a -1 message on channel score.

6.3.3.System processes

Validate_Process takes the player's moves from channel keyboard, checks that he is not trying to move off the board and limits him to one move every fifth of a second. Assuming the move is a valid one the player's new position is sent to the external world and to Monitor_Process along the f_val channel. The player's position is represented by a number between 1 and 4 which identify cells as shown below

```
1 2
3 4
```

f_cell_n carries the state changes of cell n. Each Cell_Process computes a new state at intervals defined by its period and communicates them along f_cell_n to the external world and to Monitor_Process. As mentioned in Section 6.2 the value of each cell's period is reduced after each state change until a limit is reached, the input channel rnd_n being used to generate the variations in the value of the period.

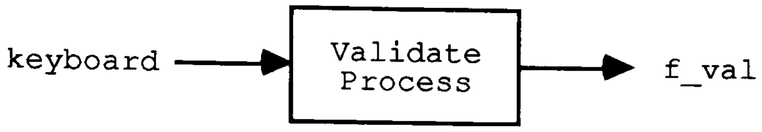
The purpose of Monitor_Process is two-fold. Firstly, it takes the cell states and the player position as inputs from channels f_cell_n and f_val and uses this information to calculate the player's score which it communicates to the external world via channel score; secondly, it detects when the player moves onto a mine cell and sends a -1 on score to indicate that the game is over.

As usual all of the processes also take a real-time clock as an input, though this is not shown on the diagram. It is assumed that in this system time is measured in milliseconds.

The next four sections outline the actual **Ruth** code which implements the processes outlined above. The complete text of the **Minesweep** program can be found in Appendix 4.

6.4 The Validate Process

6.4.1 Overview



Validate_Process is the simplest process in the **Minesweep** system. It receives the player's moves on keyboard, checks that they are valid, and if so sends the new player position to the external world and to Monitor_Process on f_val. Validate_Process will delay each move made by the player until at least 200 milliseconds after the previous move. This enforces the restriction that player moves are only responded to at fifth of a second intervals. The structure of Validate_Process is

```
Process Validate_Process                                         (6.5)
Input keyboard
Clock c
Is { Time_Check (output, c) }
whererec
output     = Validate (keyboard, start_pos, first_move) ;
start_pos  = 1 ; first_move = 0 ;

Time_Check = lambda (output, c) . ... ;
Validate = lambda (kb, pos, next_move) . ... ;

endwhererec ;          -- Validate_Process
```

Note that the single output channel from Validate_Process is enclosed between '{' and '}' to embed it in a tuple.

Validate_Process is made up of the functions Time_Check and Validate. Validate takes as its arguments the keyboard input, the player's starting position (cell 1) and the next time at which the player is allowed to make a move (initially 0). Validate checks that the player is not trying to move off the board and ensures there is a 200 millisecond gap between messages. The result of Validate is a list of [player_position,time_of_move] pairs to be sent as channel messages in f_val.

The output supervisor Time_Check takes this list as an input argument and checks it to ensure that time_of_move is late enough to be used as a messages timestamp, (i.e. that the time it is to denote has not already passed so that the message will be timed out). If time_of_move is late enough for the message to be sent this is done, if not the message is sent with time_of_move replaced by a timestamp large enough to avoid timeout. Time_Check is a simple example of defensive programming : if Validate misses a deadline Time_Check takes correcting action.

6.4.2 The Validate function

Validate passes the keyboard input onto the function Check_and_Move for checking ; the result returned by Check_and_Move being the number of the cell that the player is on after the move is completed (if the move is invalid Check_and_Move returns the current position). Check_and_Move is straightforward and will not be further discussed here; the reader is referred to Appendix 4 for details.

Function Validate is thus

```
Validate                                                    (6.6)
   = lambda (kb, pos, next_move).
       If new_pos = pos
       Then Validate (TailCh(kb), pos, move_time + interval)
       Else Cons (Cons(new_pos, move_time),
                  Validate(TailCh(kb),new_pos,
                           move_time + interval))
       whererec
       comp_delay = 5 ; interval   = 200 ;
       new_pos    = Check_and_Move (move, pos) ;
       move       = HeadCh(kb);
       move_time  = If Time (kb) + comp_delay < next_move
                    Then next_move
                    Else Time (kb) + comp_delay;
       endwhererec ; -- Validate
```

next_move is the earliest time at which the next player move can be output from

Validate_Process and is 200 milliseconds after the last output from Validate_Process. If the player attempts to move to another cell within 200 milliseconds of his last move Validate_Process will not forward the move until next_move, effectively restricting the player to one move in every 200 millisecond period.

comp_delay allows for the time between Validate_Process receiving the player's move and being able to output the new position on f_val. When specifying the **Minesweep** system 5 milliseconds were allowed for this. If, in a particular case, more that 5 milliseconds elapse so that move_time is not a valid timestamp this will be detected by Time_Check and the appropriate action taken.

It is because of this uncertainty about the validity of move_time as a message timestamp that Validate returns a list of pairs instead of a channel. If a channel were used there is a potential for it to be timed out and no more player moves would be processed. Instead the process is written defensively : channels are only constructed after their message timestamps' validity has been checked; this is done by the output supervisor Time_Check.

6.4.3. The Time_Check function

Time_Check checks each of the potential message timestamps in the list produced by Validate against the current time read from the clock c, replaces those that are too early to be used as message timestamps and constructs a channel of player positions to be sent to the screen driver and to Monitor_Process. The **Ruth** code for Time_Check is given overleaf.

```
Time_Check                                                      (6.7)
  = lambda (output, c) .
      If out_time ≥ soonest
      Then ConsCh (out_data, out_time,
                   Time_Check (Tail(output), TailClk(c)))
      Else ConsCh (out_data, soonest,
                   Time_Check (Tail(output), TailClk(c)))
      whererec
      out_time     = Tail(Head(output)) ;
      out_data     = Head(Head(output)) ;
      soonest      = HeadClk (c) + check_delay ;
      check_delay  = 1 ;
      endwhererec ; -- Time_Check
```

Note the use of check_delay to allow for the time taken after the clock is read to construct the channel and to output the message. Even when timestamps are checked as above it is impossible to guarantee that messages will be sent : there may be a longer delay than allowed for by check_delay before an attempt is made to output the message so that a timeout will occur. However the situation is not as bad as it might appear since Time_Check is a fairly small and simple function and we can be fairly certain that timeouts will not occur.

The bigger the value of check_delay the more probable that timeouts will not occur, but, on the other hand, the earlier that Validate will have to produce a result to avoid Time_Check taking its error recovery action. The compromise taken here is that Validate should produce its result with 20% of its allotted time (i.e. 1 millisecond) remaining to allow time for Time_Check to operate. Of course, this is just an informed estimate of the amount of time that will be required for Time_Check, and will be modified if it proves to be too big or too small. Estimating delay values in this way is common practice in the real-time field.

Time_Check is a **passive** output supervisor in that it waits for Validate to produce messages for output and then checks them for validity. In Time_Check messages which have missed deadlines are simply re-timestamped, though other strategies, for example discarding any incorrect messages, are obviously possible. However, all a passive output supervisor can ensure is that messages which have missed deadlines are not sent (i.e. that events do not occur at the wrong time). In many cases it is desirable to send a message to meet every deadline, even if the message does not denote the required event, and this requires an **active** output supervisor. An active output supervisor monitors the process's output, waiting for the deadline and sending a default message if the "real" message is not produced by the deadline. This ensures that an event will occur at the required time, even if the event is not the exact event required.

An active output supervisor is not required in the case of `Validate_Process` since failures to meet deadlines are not very serious : processing player input is a soft real-time problem. However it would be straightforward to provide `Validate_Process` with an active output supervisor by passing `f_val` to process `Active`.

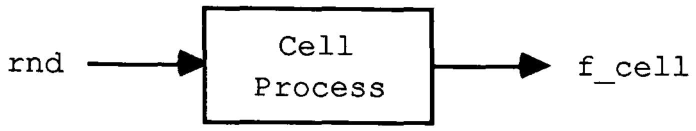
```
Process Active                                                    (6.8)
Input f_val
Clock c
Is Check(f_val, c, 0)
whererec
Check
   = lambda (f_val, c, last_output).
       If Ready (f_val, last_output + allowed_interval)
       Then ConsCh (HeadCh(f_val), out_time,
                     Check(TailCh(f_val),TailClk(c),Time(f_val)))
       Else ConsCh (default, out_time,
                     Check(f_val,TailClk(c),out_time))
       whererec
       out_time          = HeadClk(c) + check_delay ;
       default           = ... ;
       allowed_interval = 1000 ;
       check_delay       = 5 ;
       endwhererec ; -- Check
endwhererec; -- Active Process
```

The `Check` function monitors the `f_val` channel and if too great a time elapses between messages (in this case 1000 milliseconds) assumes that a timeout has occurred and sends the `default` message. Otherwise received messages are passed on unchanged, apart from having their timestamps updated.

After a timeout has occurred process `Check` simply carries on monitoring `f_val` although once a **Ruth** channel message has been timed out the channel will never contain any more messages. What we would prefer in such situations is that `Check` be able to send a message to the timed out process instructing it to reinitialise itself. Unfortunately, it is likely that any process which has had a channel message timed out has itself crashed and would thus not respond to such an instruction. The obvious solution is to simply create a new instance of the process but **Ruth** does not allow dynamic process creation. This restriction should be removed in later versions of the language and dynamic process creation is discussed further in the conclusion of this chapter and in Chapter 7.

6.5 The Cell Processes

6.5.1 Overview



Each cell process periodically calculates a new state and communicates it to the external world and to `Monitor_Process` via its `f_cell` channel. The frequency with which new states are computed depends upon the cell's period value and upon a random number input from channel `rnd`. The period value is decremented after each new state is computed until it reaches a limit of 2500 milliseconds; thus as the game progresses the cells change state faster and faster.

The states computed also depend on a random number read from `rnd`. If that value is 500 (50%) or over the cell will be a scoring one; otherwise the chance that the new state will be mine or null depends on the cell's danger value, the higher the danger the more chance that the new state will be mine. The danger value is incremented after each state change, up to a limit of 400 (40%).

```
Process Cell_Process                                          (6.9)
Input rnd
Clock c
Is { Time_Check (output, c) }
whererec
output = Cell (rnd, 0, init_state, init_danger, init_period) ;

Time_Check = lambda (output, c) . ... ;

init_state  = null ;  null = -1           ;  mine = -2        ;
init_danger = 50   ;  limit_danger = 400 ;  inc_danger = 5   ;
init_period = 20000;  limit_period = 2500;  dec_period = 250;

Cell = lambda (rnd, last_out, state, danger, period) . ... ;
Calculate_State = lambda (n,danger) ... ;
endwhererec ; -- Cell_Process
```

Cell_Process is similar in structure to Validate_Process. The list produced by Cell is checked for the validity of its time values by the passive output supervisor Time_Check. It is unlikely that timing errors will occur because the fastest a cell need ever produce messages is once every 1250 milliseconds; nonetheless the check is included for safety.

6.5.2 The Cell function

Cell takes as arguments the channel rnd, last_out, the time of the last state change, the current value of the state and the current values of danger and period and produces as its result a list of [state,time_of_change] pairs.

```
Cell                                                                    (6.10)
  = lambda (rnd, last_out, state, danger, period).
     If new_state = state
     Then Cell  (TailCh(TailCh(rnd)), out_time, state,
                 new_danger, new_period)
     Else Cons  (Cons(new_state, out_time),
                 Cell  (TailCh(TailCh(rnd)), out_time, new_state,
                        new_danger, new_period))
     whererec
     out_time    = last_out + period +
                    ((HeadCh(rnd) - 500) * period) / 1000 ;
     new_state   = Calculate_State(HeadCh(TailCh(rnd)),danger) ;
     new_period = If (period - dec_period) ≤ limit_period
                   Then limit_period
                   Else period - dec_period ;
     new_danger = If (danger + inc_danger) ≥ limit_danger
                   Then limit_danger
                   Else danger + inc_danger ;
     endwhererec ; -- Cell
```

The Cell function uses the Calculate_State function to produce the new states and builds them into a list of [state,time of change] pairs. Calculate_State is straightforward and will not be discussed further; the reader is referred to Appendix 4 for details. Note that both the time of the next state change and the new state depend on a random number input. Note also the limiting of the values of the danger and period to limit_danger and limit_period respectively.

6.5.3 The Time_Check function

The Time_Check function used in the cell processes is almost identical to that used in Validate_Process; the only difference being the action taken when a timing error is encountered.

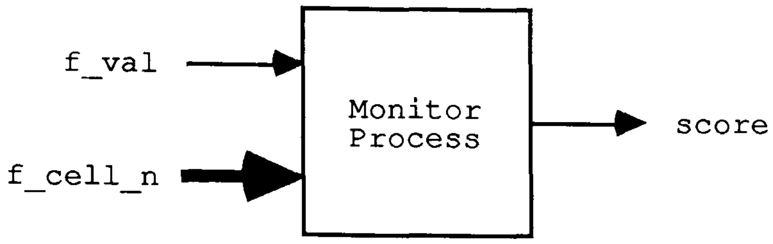
```
Time_Check                                                      (6.11)
  = lambda (output, c) .
    If out_time ≥ soonest
    Then ConsCh (out_data, out_time,
                 Time_Check (Tail(output), TailClk(c)))
    Else Time_Check (Tail(output), TailClk(c))
    whererec
    out_time      = Tail(Head(output)) ;
    out_data      = Head(Head(output)) ;
    soonest       = HeadClk (c) + check_delay ;
    check_delay   = 250 ;
    endwhererec ; -- Time_Check
```

Whenever the Cell function defines a state change to occur at a time that is too early for Cell_Process to be sure of sending the message in time the state change is simply discarded. Since it is always possible for the cell's new state to be the same as its old state this is a safe strategy for error recovery.

The smallest interval between two successive state changes is 1250 milliseconds and consequently we would expect timing errors to be infrequent. In Validate_Process 20% of the available computation time was assumed to be reserved for the Time_Check function; here 20% of the minimum available time is allocated to Time_Check and thus check_delay is 250 milliseconds. As with the check_delay value used in Time_Check in Validate this value is an informed estimate based on real-time domain experience.

6.6 The Monitor Process

6.6.1 Overview



Monitor_Process takes as input the player's moves from Validate_Process (via f_val) and the state changes from each each Cell_Process (via the f_cell_n channels). Each time the player moves it calculates his new score and, if the score has changed, sends the new score along channel score. If the player moves onto a mine cell Monitor_Process informs the external world by sending -1 along score.

```
Process Monitor_Process
Input f_cell_1, f_cell_2, f_cell_3, f_cell_4, f_val
Clock c
Is { Time_Check (output, c) }
whererec
output = Monitor (inputs,init_state,init_player_pos,init_score) ;
inputs = Scan_and_Sort (f_cell_1, f_cell_2, f_cell_3, f_cell_4,
                        f_val, init_scan) ;

init_score      = 0   ; init_player_pos = 1   ;
init_scan       = 200 ; scan_interval   = 200 ;
mine            = -2  ; null            = -1  ;

init_state = Cons(null,Cons(null,Cons(null,Cons(null,Nil)))) ;

Time_Check    = lambda (output, c) . ... ;
Scan_and_Sort = lambda (f_cell_1, f_cell_2, f_cell_3, f_cell_4,
                        f_val, scan_time) . ... ;
Monitor       = lambda (inputs, state, player_pos, score) . ... ;
endwhererec ; -- Monitor_Process
```

Monitor_Process operates in much the same way as Validate_Process and

Cell_Process in that it uses a passive output supervisor Time_Check to check a list of [score,time] pairs before converting them into channel messages. The input channels to Monitor_Process are timewise merged into the list inputs by the function Scan_and_Sort; thus the earlier a message arrives at Monitor_Process the earlier it appears in inputs. This simplicity in ensuring that events are processed in the order they occur is a major advantage of the explicit timestamps used by **Ruth**.

The list of [score,time] pairs is produced by the function Monitor which takes as its arguments the list inputs, the state of the board (initially all the cells are null), the player's current position (initially on cell 1) and the current score (initially 0). Monitor works down the inputs list processing each message, and thus each event in the system, in the order in which they occurred. Each cell state change message results in Monitor updating its state value; each player move results in the score being updated if the player moved onto a scoring cell, the sending of -1 if the player moved onto a mine, and no message at all if the cell is null.

6.6.2 Getting the inputs

The Scan_and_Sort function operates in exactly the way its name implies. Periodically it scans all the input channels and builds a list of all the messages that arrived since the last scan. Scan_and_Sort assumes that only one message can have arrived from each of the other processes since the last scan. In order to ensure this Scan_and_Sort must scan the channels at least every 200 milliseconds. This is the smallest possible interval between messages on f_val. None of the other channels will produce messages as fast as this so this is the limiting case.

The general principle behind Scan_and_Sort is that all the input messages available at this scan_time are collected into an event_list which is then sorted by order of occurrence of the events they denote (earliest first). The resulting list is appended to the list resulting from recursively applying Scan_and_Sort to the input channels with the messages already received removed. Thus the result of Scan_and_Sort is a list containing all the events reported to Monitor_Process in order of their occurrence : a timewise merge of its inputs.

```
Scan_and_Sort                                                    (6.12)
  = lambda (f_cell_1, f_cell_2, f_cell_3, f_cell_4, f_val,
           scan_time) .
      Append(Quicksort(event_list, Nil),
            Scan_and_Sort(new_f_cell_1,new_f_cell_2,
                          new_f_cell_3,new_f_cell_4,
                          new_f_val,scan_time + scan_interval))
      whererec
      event_list
       = Get_Events ({f_cell_1, f_cell_2, f_cell_3,
                     f_cell_4, f_val}, 5, scan_time)) ;

      new_f_cell_1 = New_Chan (f_cell_1, scan_time) ;
      new_f_cell_2 = New_Chan (f_cell_2, scan_time) ;
      new_f_cell_3 = New_Chan (f_cell_3, scan_time) ;
      new_f_cell_4 = New_Chan (f_cell_4, scan_time) ;
      new_f_val    = New_Chan (f_val,    scan_time) ;
      endwhererec ; -- Scan_and_Sort
```

Here `Append` is the usual function for appending two lists and `Quicksort` performs a quicksort on the `event_list` according to the timestamps of the messages. Function `New_Chan` simply returns the **`Tailch`** of its channel argument if it is **`Ready`** at the time denoted by its second argument. Thus it discards all messages which have been incorporated in `event_list` in this scan. For the text of these functions the reader is referred to Appendix 4.

Because all the events are combined into one list they must be tagged to show their origin. Thus every element of `event_list` is a pair comprising an origin tag and the event itself. To tag the events we use the index of their origin channel in the list of inputs to `Monitor_Process`, starting at 1. Thus `f_val` events are tagged 5. Note that the event itself is a pair whose first element is a new cell state or the cell that the player has moved to, and whose second element is the timestamp of the message. The tagging is performed by the function `Get_Event` when it reads the messages from the input channels.

```
Get_Events                                                    (6.13)
  = lambda (chs, n, t) .
     If n < 1
     Then Nil
     Else If Ready(chs!n, t)
          Then Cons(Cons(n, event), Get_Events (chs, n-1, t) )
          Else Get_Events (chs, n-1, t) ;
     where
     event = Cons(HeadCh(chs!n), Time(chs!n) ) ;
     endwhere ; -- Get_Events
```

Get_Events collects the messages which have arrived since the last scan; Quicksort sorts these messages into order of arrival. Since there are so many input channels it is clearer and simpler to separate these two operations from each other than to combine them into one function as is done be the timewise merge functions seen earlier.

6.6.3 The Monitor function

The Monitor function works down the list of events inputs, processing the events in turn and updating the score and state of the cells in accordance with each event.

```
Monitor                                                        (6.14)
= lambda (inputs, state, player_pos, score) .
   score_out
   whererec
   event      = Head(inputs) ;
   event_type = Head(event) ;
   event_data = Head(Tail(event)) ;
   event_time = Tail(Tail(event)) ;
   out_time   = event_time + comp_delay ;
   comp_delay = 50 ;

   score_out = ... ;

   endwhererec ; -- Monitor
```

Monitor takes the first event in the list of events inputs and decomposes it into event_type, event_data and event_time. event_type is the tag attached to the event by Scan_and_Sort and so denotes the origin of the event message. event_data is the data part of the message denoting a new position if the event came from the player and a new state if the event originated from a cell. event_time is the time the event occurred.

The updated score is output at out_time which is 50 milliseconds after the event actually occurred. This allows for the time involved in processing the message within Monitor_Process, though if 50 milliseconds proves to be too short a time the output supervisor will trap the error.

score_out is calculated as follows

```
score_out = If Lookup(new_state,new_player_pos) = mine     (6.15)
            Then Cons (Cons(-1,out_time), Nil)
            Else If new_score ≠ score
                 Then Cons (Cons(new_score,out_time), rest)
                 Else rest ;  -- score_out

rest = Monitor(Tail(inputs),new_state,new_player_pos,new_score) ;

new_score = ... ;
new_player_pos = ... ;
new_state = ...  ;
Lookup = lambda (table, n). ... ;
```

Here new_score is the the updated value of score, new_player_pos is the new player position and new_state is the new cell state list. If the event was a player move then new_state will have the same value as state; if the event was a cell state change then new_score and new_player_pos will have the same values as score and player_pos respectively. Lookup returns the n^{th} element of the list table. These definitions are straightforward and the reader is referred to Appendix 4 for details.

Note that if the player lands on a mine, so ending the game, the pair [-1,out_time] is produced followed by Nil. Since the game is over Monitor processes no more events.

6.6.4 The Time_Check function

The output supervisor for Monitor_Process is similar to that for the other two process, the only difference once again being its behaviour in the advent of timing errors.

```
Time_Check                                                    (6.16)
  = lambda (output, c) .
      If out_time ≥ soonest
      Then ConsCh (out_data, out_time,
                   Time_Check (Tail(output), TailClk(c)))
      Else ConsCh (-1, soonest, stop)
      whererec
      out_time     = Tail(Head(output)) ;
      out_data     = Head(Head(output)) ;
      soonest      = HeadClk (c) + check_delay ;
      stop         = ConsCh (-1, 0, stop) ;
      check_delay  = 10 ;
      endwhererec ; -- Time_Check
```

When a timing error is detected Time_Check terminates the game. A more sophisticated strategy, such as sending the score as soon as possible in a similar way to Validate_Process, is obviously possible. However, since Monitor_Process is the most important process in the **Minesweep** system we choose to regard any timing failures within it a fatal errors. The most useful course of action would be to pre-emptively reinitialise the whole system but **Ruth** provides no primitive for pre-empting processes.

Instead Monitor_Process produces the -1 message, so ending the game, and ceases to process messages. There is no elegant way of terminating a channel in **Ruth** since it is slightly unusual for a real-time system ever to terminate. The only way to achieve the desired effect is with the recursive definition of stop above which will produce a timeout error since it attempts to use a zero timestamp. Note that 20% of the available computation time (i.e. 10 milliseconds) is reserved for Time_Check.This ensures, as far as is possible in a real-time system, that the -1 message will not be timed out but will be received by the external world. Once again, the value chosen for check_delay is an informed estimate and may change as a result of experiences running the implementation.

The total delay between messages reaching Monitor_Process and the production of a message in response is 50 milliseconds. Thus when the player makes a move any change in his score will not be visible to him for 55 milliseconds : a 50 millisecond delay in Monitor_Process and a 5 millisecond delay in Validate_Process. As far as the player is concerned his moves occur when he presses one of the 'u', 'd', 'l' or 'r 'keys; as far as Monitor_Process and the external world are concerned the player actually makes his move 5 milliseconds after a key is pressed, and he is not credited with any score for the move for a further 50 milliseconds. However, as far as the user is concerned, cell state changes occur when they appear upon the screen and thus, to the user, there seems to be an unfair delay in processing his moves.

A partial solution to this problem would be to delay cell state changes by 5 milliseconds in the Scan_and_Sort function. Thus the state of the board held in Monitor would always be consistent with a 5 millisecond delay on both the player moves and the cell state changes and the player would not find himself moving onto a mine cell which was not a mine when he made the move. Unfortunately this would be inconsistent with what actually appeared on the screen as a cell would appear to change state 5 milliseconds before the new state had any affect on the progress of the game.

There is no total solution to this problem since it is caused by the player's perception of reality : a cell changes state when the new state appears on the screen; the player moves when he presses a key. Such problems with the perception of when events occur are common to real-time systems and derive from the fact that it takes time for a computer to process its inputs and produce its outputs. There is no way that a real-time program can ever have a totally up to date picture of the external world and this is something that the real-time programmer must learn to live with.

This completes the definition of the **Minesweep** implementation. To see how it all fits together the reader is once again referred to Appendix 4 where the full text of the implementation is given.

In the next, concluding, section of this chapter we shall examine and evaluate our solution to the problem of implementing the **Minesweep** specification.

6.7 Conclusion

The purpose of this chapter has been to evaluate the suitability of the language **Ruth** for implementing real-time systems by applying it to a substantial real-time problem. The problem chosen was the real-time computer game **Minesweep**. Computer games exhibit fairly complex real-time behaviour but have the advantage of requiring no specialist knowledge to implement and/or explain.

The emphasis has been upon the real-time problems posed by **Minesweep** and the suitability of **Ruth** for expressing solutions to these problems. Consequently we have abstracted away from several non real-time issues : for example how the screen is to be driven and where the random number inputs are generated.

It could well be argued that we have been somewhat pedantic in specifying real-time behaviour in the **Minesweep** program. A computer game interfaces to a human as opposed to machinery and human reaction times are very slow in comparison to a machine's. (i.e. seconds rather than milliseconds). Thus a real implementation of **Minesweep** could probably dispense with the `Quicksort` function in `Scan_and_Sort` and simply treat all the events detected in a scanning phase as happening simultaneously. This strategy would, however, lead to several discrepancies in the real-time behaviour of the system. For example `Monitor_Process` could receive two messages in a scan : the player moves off cell n, and cell n changes state from a scoring cell to a mine. If both events are assumed to occur simultaneously then the game is over since the player has landed on a mine. However the player could have moved onto the cell before the cell changed state in which case his score should be increased by the relevant amount.

Although it is very unlikely that the player would notice any discrepancies in the real-time behaviour of a **Minesweep** implementation given the very small (to a human) time periods involved, in a narrow real-time system such discrepancies would be "noticed", and could have disastrous effects. To maintain consistency of real-time behaviour requires that the real-time language used be able to detect exactly when events occurred or did not occur, and the ability to specify exactly when events should occur : in other words time expressibility. By checking the timestamps on channel messages using Ready and Time the **Minesweep** program maintains the consistency of its real-time behaviour.

The **Minesweep** program also exhibited the use of defensive programming techniques in **Ruth** : each process's output was checked by the Time_Check passive output supervisor. However a passive output supervisor is not a total solution to the defensive programming problem since it can only prevent incorrect messages being output; it cannot output default messages to ensure that deadlines are always met. Although it is possible to write active output supervisor processes in **Ruth** there is no way in the functional framework for such a process to preempt a rogue process and force it to take corrective action. The obvious solution is simply to create a new version of the process and ignore the rogue but **Ruth** does not allow dynamic process creation. In the next chapter we shall look at a way in which this situation could be improved.

Ruth's insistence on the explicit naming of all channels and processes in a configuration causes problems with handling large numbers of processes and channels. This led us to restrict the **Minesweep** board to only four cells.

The handling of channels within the language generally is definitely an area which could be greatly improved. The only data structure available for use with channels is the tuple and a tuple can only be constructed by listing all its elements. There is no operation like Cons on s-expressions which would allow us to add channels to tuples and thus we cannot write recursive functions that "map" down tuples to produce other tuples . This restriction was particularly felt in Monitor_Process, for example in the separate definitions of new_f_cell_1 through new_f_val in Scan_and_Sort.

The problems mentioned above are largely peripheral to the central goal of **Ruth** which was to prove that there is no reason why a purely functional language could not be used for writing real-time systems. The **Minesweep** program suggests that we have succeeded in this aim. Although certain features of the language could be improved those dealing with specifically with real-time (channels, clocks and their associated operations) are more than adequate for dealing with real-time systems.

In the final chapter of this thesis we shall make a fuller review of what has been achieved in this work. In particular we will consider future directions for the work, both in terms of language improvements and theoretical issues.

Chapter 7 : Review, Assessment And Future Work

7.1 Review

7.1.1 What is a real-time system ?

A real-time problem can be classified as one in which *when* events occur is as important as *what* events occur. A further refinement can be made along two axes : the amount of time a system has to react to an event and the scale of damage if these deadlines are not met. The importance of when things happen in a real-time system has two major implications : the system must specify when it wishes events to happen and it must be able to detect when events have happened. Thus, any language used for the implementation of real-time systems must have what we have called time expressibility : the ability to express facts about time.

The timing constraints inherent in real-time programming also have a major impact on the type of algorithms that are used. Real-time programmers tend to program in a very defensive manner : if there is the slightest doubt as to whether a program will meet its constraints then it is rewritten. In most real-time systems the time-critical part of the software can be expected to complete its tasks in as little as half the time available to it. Even so, individual processes in a real-time system are usually written in such a way that the failure of another process to meet its time constraints can be recovered from. This usually involves writing a large amount of error handling code which (hopefully) is never executed.

7.1.2 Current approaches to real-time language design

In Chapter 2 we looked at three different approaches to real-time language design, imperative, dataflow and functional, and examined their relative merits in terms of time expressibility and defensive programming. As a result of this survey two general types of approach were identified : the pragmatic, defensive approach exemplified by languages such as Ada, and the theoretical, optimistic approach exemplified by LUSTRE.

The defensive languages assume nothing about the real-time characteristics of a particular language implementation and give no guarantees that deadlines will be met. Instead facilities are provided to support defensive programming : software can monitor its own progress in time and take recovery action if deadlines are not met.

The optimistic languages assume a particular real-time behaviour from language implementations. For example LUSTRE assumes the strong synchrony hypothesis : machine operations take negligible time and deadlines are always met. Thus LUSTRE provides no support for detecting and recovering from timing errors; the focus is upon specifying the temporal behaviour required rather than whether that behaviour can be achieved. The other optimistic language considered, ART, assumes merely that the exact time taken for each machine operation is known and thus that temporal behaviour can be predicted from the semantics of the language. Although this is not as strong an assumption as strong synchrony it may well be equally unrealistic for many, more complex, language operations; also, determining the real-time behaviour of a program requires that the complete program be proved from the semantics which is a lengthy and potentially error prone task at present.

Given that the strong synchrony hypothesis is unrealistic when working with real-time systems, and that the current state of the art of program proving is not generally practical, we are forced onto the defensive. The basic principles of the language **Ruth** are thus those of a defensive language : nothing is assumed about an implementation's real-time characteristics and the language provides facilities for monitoring progress in time and detecting and recovering from timing errors.

7.1.3 The language **Ruth**

A **Ruth** program comprises a static set of processes communicating via streams of (numerically) timestamped atomic messages, in which each message denotes an event in the system. Such streams are called channels in **Ruth** and each process is a function mapping a tuple of input channels to a tuple of output channels.

The timestamp on a channel message defines when the message is required to arrive at its intended destination (either another process or the external world) so giving the **Ruth** programmer the ability to specify when events happen. For simplicity, we have assumed no communications delays on channels and no clock skewing between processes.

A process also takes a real-time clock stream as input. As the program executes the values in the clock stream are (lazily) instantiated with the current value of real-time, thus allowing the programmer access to real-time information. This approach was first suggested in [Burton 88].

In order to detect the occurrence of an event **Ruth** supplies an operation for testing the timestamp of the first message in a channel against an integer value : the **Ready** test. **Ready** is a predicate which returns `true` if the timestamp of the first message in the channel is less than or equal to the number being tested against and `false` otherwise. The most important requirement of **Ready** is that it should be non-blocking , that is, that the event need not have happened (there need be no message in the channel) for **Ready** to return a result. In real-time systems the non-occurrence of an event by a particular time carries (almost) as much information as its occurrence and **Ready** must be able to detect this situation.

7.1.4 Herring-bone domains and clock-driven timing

To give a formal definition of operations such as **Ready** requires that the semantic model used be able to express facts about time. The herring-bone domains used in [Broy 83] to give the semantics of ART are just such a model and they were used to provide a framework for the semantic definition of **Ruth**.

When dealing with the semantics of real-time programming languages one of the most important things that must be defined is when expressions produce results. In the semantics of **Ruth** these times depended upon values read from a clock, an approach we called clock-driven timing. Clock-driven timing gives a higher level of abstraction than using fixed durations for each machine operation (delay-driven timing) since there is no need to fix such things as evaluation orders. Yet it more accurately models the real world than assuming no delays at all (data-driven timing). Moreover, by constraining the values read from the semantic clocks in certain ways data dependencies can be expressed (e.g. the result of an addition cannot be available until after the two numbers to be added are computed) without defining low level implementation details.

7.1.5 Conclusion

The next section of this chapter assesses the merits and failings of **Ruth** as a real-time programming language and suggests several improvements. Section 7.3 outlines a prototype implementation for the language. In Section 7.4 the focus is upon theoretical issues and in particular upon what kinds of formal reasoning and transformation are possible with real-time software. Based on this Section 7.5 indicates a possible direction for future work.

7.2 Ruth As A Real-Time Programming Language

7.2.1 Time expressibility

Our primary goal with **Ruth** was to produce a purely functional programming language for writing real-time software. The essential feature of a real-time language is its time expressibility, that is, the data types for representing temporal information and the operations on those data types. In **Ruth** time is represented by the integer values of channel timestamps and clock times. In general this method has proved highly effective : channel timestamps allow the **Ruth** programmer to detect when events happen and to specify when events should happen in a simple and elegant way. By treating a real-time clock as an input stream **Ruth** allows access to the current time in a purely functional manner.

One restrictive feature of the channel/timestamp model is the fact that a programmer must always specify a timestamp even when he does not care when the message is delivered, provided that it will be delivered eventually. In real-time systems there is often a need to handle "don't care" outputs, for example the logging of errors in an engine control system. The only timing restriction is that all errors should be logged "eventually".

[Harrison 87] proposed that "don't care" output could be expressed by supplying a complete clock as the second argument to `ConsCh`. Operationally the intention was that the clock should be instantiated at the moment the message is sent, thus avoiding timeouts. This was abandoned because there seemed to be no way of providing a semantic definition for it within the semantic model being used. According to this model a program is supplied with clock tree as an input and nothing can be assumed about the values contained in that tree.

A better approach is simply to allow **Ruth** processes to produce lazy streams of (untimestamped) atoms as output as well as channels. No timestamps are specified since the programmer does not care when the atoms in the streams arrive at their destinations.

This has the disadvantage that a receiving process could not use `Time` or, more importantly, `Ready` on such an input stream. The view taken here is that if the time at which a particular atom is produced is of no importance to the producer of the atom then it has no importance to the consumer. If this is not the case then a channel should be used.

If a **Ruth** process fails to meet a time deadline this is treated as a fatal error. In other words, if an attempt is made either to construct a channel with a message timestamp that is less than the current value of time, or to send an out of date channel message to another process, then a timeout occurs and the $<\infty, \perp>$ element of $\mathbb{CHAN}$ results.

An alternative approach would be to replace the erroneous message with a "time fault" message as is done in real-time Lucid. This has the obvious advantage that the process which produced the late message is not assumed to be fatally flawed and may go on to produce further (hopefully) on-time messages. The only disadvantage is that we cannot predict from the semantics when a message will be replaced by a time fault. In a channel construction such replacement depends upon when the channel is constructed and at a process boundary it depends upon when the attempt is made to output the message. In the semantics both of these times are read from the semantic clock and we have no information about the values in clocks. On the other hand the robustness of **Ruth** programs would be markedly improved by the introduction of time faults since the process producing the time fault could be informed of this by the process receiving it, and could thus take correcting action. For this reason time faults are a necessary addition to **Ruth**.

A problem which which occurred when writing the **Minesweep** program was that of finite channels. Normally a real-time program is never expected to terminate, but to go on producing results "for ever" and consequently it was felt there was no need to consider a means for signalling the end of a channel. Even if all real-time programs never terminate, and the **Minesweep** program shows that this is not true, there is still a use for a channel terminator, particularly if **Ruth** were to allow dynamic process creation (see below).

To allow the programmer to terminate a channel **Ruth** should provide a primitive such as `End_Chan` below

```
End_Chan (n) = <n, End>                                    (7.1)
```

and the definitions of `HeadCh`, `TailCh`, `Time` and `Ready` would be augmented in the obvious way.

7.2.2 Defensive programming and dynamic process creation

As was seen in the **Minesweep** program it is simple to write passive output supervisors in **Ruth** to support defensive programming. However passive output supervisors have an important weakness : they can only react to incorrect message timestamps to prevent timeouts, they cannot send messages to ensure deadlines are met. The latter requires an active output supervisor.

In **Ruth** an active output supervisor would be written as a process which acts as a watchdog on a producing process's output channel(s). When a deadline is not met a default message would be inserted. Once such a situation has occurred it is frequently the case that some form of re-initialisation of the producing process is required. Since there is no way to preempt a process in a functional language, and thus force it to reinitialise, the obvious technique would be to create a new version of the producing process and rely on the underlying implementation to "garbage collect" the original. Unfortunately dynamic process creation is not allowed in **Ruth** so this strategy cannot be used. This is a serious weakness of **Ruth**.

The process structuring facilities in **Ruth** were influenced strongly by those of occam. This was largely for reasons of simplicity since our primary concern was the features of the language supporting its time expressibility and not the process structure. Consequently a **Ruth** program is a static configuration of processes, and processes are distinct from functions.

In retrospect there seems no reason why a process should not be treated as a function which produces tuples of channels as results. The language semantics already treats processes in just this way. Instead of a **Ruth** program being a configuration of processes it would be itself a process whose process expression was allowed to define and apply other processes.

7.2.3 Clock skewing and communication delays

Throughout this work we have made two simplifying assumptions which are not justified in the real world. Firstly, that the real-time clocks on different physical processors will always be synchronised (i.e. there will be no clock skewing); and secondly, that channel communication is instantaneous. Before **Ruth** could be used on real world problems these assumptions must be discharged.

Clock skewing between processors can be dealt with at the implementation level via methods such as Lamport's algorithm [Lamport 78] mentioned in Chapter 3. Any realistic implementation of **Ruth** on a distributed set of processors would have to use such an algorithm.

Channel communication delays are slightly more complex to handle. Unpredictable communication delays are a fundamental problem in real-time programming to which there seems no obvious solution. Once again it seems that the real-time programmer is forced onto the defensive. Programs must be written in such a way that messages will reach their destinations on time in all but the worst situations. When a message does fail to meet its time deadlines a real-time program must be able to recover.

It should be noted that the use of the `filter` function in the definition of process application could be interpreted as modelling unpredictable communication delays. `filter` uses the clock values to denote the time that messages become available at their destinations. Although we have assumed that this is the same as the time at which the message leaves its source this assumption has no bearing on `filter` which reads the time values it checks against directly from a semantic clock about whose values we have no information. It could equally well be assumed that the times read from the clock denote only the arrival time of the message, which is strictly greater than the sending time.

7.2.4 Conclusion

In this section we have looked at **Ruth** as a programming language and have found it wanting in several respects. For **Ruth** to be regarded as useful in the real world the improvements mentioned in this section, and possibly several others, would have to be made. It should be noted however that we have found **Ruth**'s underlying notion of real-time systems as a set of real-time processes communicating via streams of timestamped messages to be a simple, powerful and elegant model. This model forms the basis of the language **STRuth** which is introduced in Section 7.5.

7.3 Outline Implementation

This section outlines a possible prototype implementation for **Ruth**. The intention is to show that there is no conceptual difficulty in producing an implementation for **Ruth**; no implication that this approach is the optimal one is intended.

7.3.1 Overview

A **Ruth** program is a configuration of several processes executing in parallel. We propose that each process should be mapped onto one of a set of processing agents which will execute it. Each agent thus has two tasks : to execute its **Ruth** process and to communicate the process's results to other agents and to the outside world. We shall assume that agents execute **Ruth** processes via lazy SECD machines ([Landin 64], [Henderson 80], [Henderson et. al. 83]). Channels are implemented as head-strict streams of `<timestamp,atom>` pairs and the **Ruth** SECD machine is extended to allow for multiple I/O channels along the lines of [Jones 84a, 84b].

The system will be implemented in occam and will run on one transputer [INMOS 87]. Thus each agent (and therefore each **Ruth** process) is implemented as one occam process. The implementation is restricted to one transputer to avoid the problem of clock skewing and message transmission delays.

Agents will communicate via occam channels; each occam channel will carry one **Ruth** channel. We shall refer to these occam channels as pipes. occam channels may be either *soft* channels linking different occam processes on the same chip via on-chip memory locations, or *hard* channels linking occam processes with the external environment via the transputer's communication links. Channels between **Ruth** processes will be implemented as soft occam channels; channels between **Ruth** processes and the external environment will be implemented as hard occam channels.

A **Ruth** channel has exactly one source process, though it may be consumed by any number of destination processes. However, since channels are implemented as occam channels, and since an occam channel has exactly one source and one target process, this prototype implementation imposes the added restriction that each **Ruth** channel may only be consumed by one destination process.

An agent executes a **Ruth** process via a series of transitions, each of which has two phases : an execution phase and a communication phase. In the execution phase the agent will perform an SECD machine transition and in the communication phase it will send and receive messages to and from other agents and the external environment. Agents perform this cycle asynchronously; two agents cannot transfer messages between themselves unless both agent is in its communication phase.

The state of an agent can be represented as a tuple of the form

```
A : [M, I, O, T]                                              (7.2)
```

Here A is the agent number (1 ≤ A ≤ Number of agents). M is the SECD machine executing the **Ruth** process. I is a list of lists, one list for each input pipe; each list contains messages that have been received by the agent but not yet read by the **Ruth** process. O contains an entry for each output pipe; at the end of an execution phase each entry contains the timestamped message (if any) produced by the last execution phase for output on that output pipe. Messages are sent to their destinations in the communication phase immediately following the execution phase that produces them, and are then removed from O. Consequently, at the end of a communication phase there will be no messages stored in O. The receiving agent has the responsibility for storing messages until they can be dealt with. Finally, T is the agent's integer value for the current time; T is incremented at the end of each communication phase from the processor's hardware clock by amounts which depend on that clock (and thus on the speed of the implementation).

The working of the execution phase is essentially identical to that given in [Jones 84a]. Three points should be noted :

(i) As in any functional language implementation the cell space must be allocated dynamically and periodically garbage collected. It would be possible to use traditional mark-sweep garbage collection on an agent's cell space but this would produce unpredictable (and fairly large) delays on agent operations which is highly undesirable in a real-time system. Instead an incremental, copying method such as that of [Baker 78] or the more efficient [Liebermann and Hewitt 83] will be used.

(ii) Instead of performing I/O directly with the external environment an SECD machine executing a **Ruth** process will perform I/O via it's agent's I and O registers. Instead of reading input directly from I/O devices the SECD machine reads messages from the I register. Thus, the SECD machine instruction INPUT is modified to read its result from the relevant I register entry, instead of directly from the external world. Similarly, instead of sending output directly to I/O devices the SECD machine instruction OUTPUT sends messages to the relevant O register.

The INPUT instruction is a blocking input : if there are no messages in the relevant I register when the SECD machine executes an input instruction then the execution phase terminates and the communication phase is entered. When the communication phase ends the INPUT instruction is re-tried.This cycle continues until a message is put in the relevant I register. The OUTPUT instruction always succeeds in placing its argument in the relevant O register.

(iii) Program clocks are implemented as head-strict streams of integers; the values in the stream are instantiated with the value of T whenever the SECD machine M executes a HEADCLK instruction.

Apart from a brief discussion in Section 7.3.3 on the implementation of the **Ready** test the execution phase will not concern us further here. For further details the reader is referred to [Jones 84a, 84b].

7.3.2 The communication phase

The input and output registers are of the form

$$\begin{aligned} O &= ((p^1,m^1) \ \ldots \ (p^k,m^k)) \\ I &= ((p_1,l_1) \ \ldots \ (p_j,l_j)) \end{aligned} \tag{7.3}$$

Each (p^i,m^i) pair in the list O denotes that the timestamped message m^i is to be sent along the pipe p^i in the next communication phase. If there is no message to be transmitted for an output pipe p^i then m^i is `Nil`. Each (p_i,l_i) pair in the list I denotes that the list of timestamped messages l_i have been received along the pipe p_i and have not yet been read by the agent's SECD machine, M. If there are no messages received from p_i which have not been read by M then l_i is `Nil`.

Note that it is the responsibility of the receiving agent to buffer messages if a sending agent is producing messages faster than a receiving agent wishes to read them. Thus I must store a list of messages for each input pipe. O need only store a single message per output pipe since each execution phase cannot produce more than one message per channel (see [Jones 84a] for details), and these messages are immediately forwarded to their destinations.

A further responsibility of a receiving agent is to check that the timestamps on incoming messages are not out of date. Each of the messages sent to an agent A is checked against the current time T when it is received. If a message has a timestamp less than than the value of T then it is out of date and a timeout error has occurred. Instead of appending the message to the relevant l_i an error value is appended indicating that a timeout has occurred. All further messages in this pipe will be ignored by agent A.

When giving the semantics of **Ruth** the check on message timestamps is performed by an application of the `filter` function in the sending process. In this prototype implementation the check is actually performed in the receiving process. This allows for easier handling of communication delays in future multi-processor implementations and is no more complex than performing the check in the sending process.

To give a flavour of how channel communication proceeds in the communication phase consider two agents A_1 and A_2. At the start of a communication phase A_1 wishes to send the message d with timestamp t along the pipe p to A_2. For simplicity, assume that A_1 is executing a **Ruth** process with only one output channel, and thus that p is the only pipe in A_1's output register. Similarly, assume that the **Ruth** process being executed by agent A_2 has only one input channel, and thus, that p is the only pipe in A_2's input register. We shall also assume that A_1 receives no input messages during this communication phase and that A_2 sends no output messages.

At the start of the communication phase the states of A_1 and A_2 are

$$A_1 : [M_1,\quad I_1,\quad (\ (p,(t,d))\ ),\quad T_1] \qquad (7.4)$$

and

$$A_2 : [M_2,\quad (\ (p,l)\ ),\quad O_2,\quad T_2]$$

Here l is the list of messages which A_2 has received on pipe p which have not yet been read by the **Ruth** process being executed by M_2. If T_2 is no later than t then the message (t,d) is appended to l since it has arrived at A_2 in time. Otherwise l records that (t,d) is an out of date message by appending `error` to the message list for p in A_2. Any further messages received on p will be ignored so that the **Ruth** process being executed by A_2 will receive no more messages from A_2 after the timeout.

Thus, at the end of the communication phase, the agent's states are

$$A_1 : [M_1,\quad I_1,\quad (\ (p,\mathrm{Nil})\ ),\quad T_1+\delta_1] \qquad (7.5)$$

and

$$A_2 : [M_2,\quad (\ (p,\mathrm{append}(l,(t,d)))\ ),\quad O_2,\quad T_2+\delta_2] \text{ iff } T_2 \le t$$

$$A_2 : [M_2,\quad (\ (p,\mathrm{append}(l,\mathrm{error}))\ ),\quad O_2,\quad T_2+\delta_2] \text{ otherwise}$$

```
where
append
 = lambda (l,m) .
    If l = error
    Then error
    Else If IsNil(l)
         Then Cons(m, Nil)
         Else Cons(Head(l), append(Tail(l), m))
```

T_1 and T_2 are incremented by δ_1 and δ_2 respectively, thus modelling the passage of time as execution proceeds. δ_1 and δ_2 will probably not be the same since the time A_1 takes to send the message and remove it from O will probably be different to the time taken by A_2 to receive the message and store it in I.

This simple case generalises in a fairly straight forward way to cases involving several messages and agents.

As mentioned above both A_1 and A_2 must be in a communication phase for a message to transferred between them. An attempt to send a message from an O register blocks an agent : A_1 cannot proceed with its next execution phase until A_2 has received the (t,d) message. Thus A_1 must wait for A_2 to enter its communication phase. An attempt to input a message to an I register does not block and agent : A_1 can simply scan for inputs and if none are available it proceeds. The INPUT SECD machine instruction ensures that an attempt by a **Ruth** process to read a message from a channel will always be blocked until a message has arrived. An agent must, however, be able to scan for input messages in a non-blocking manner so that the **Ready** test can be successfully implemented (see Section 7.3.3 below). In the worst case a source agent may enter its communication phase and try to send a message just after the destination agent has completed its scan. The source agent will be delayed through the rest of the destination agent's communication phase and the following execution phase before it can send the message and proceed.

An agent must scan its inputs and send its outputs in parallel to avoid deadlock. For example, if agent A_1 attempts to send a message to agent A_2 and A_2 attempts to send a message to A_1. Both agents are attempting to send messages and thus, unless they are scanning for inputs in parallel with this, deadlock will occur.

7.3.3 Implementing the **Ready** test

As mentioned above, an SECD machine executing a **Ruth** process performs all its I/O with its agents I and O registers. To implement the **Ruth Ready** test requires the addition of an extra instruction, READY, to the SECD machine given in [Jones 84a, 84b]. The READY instruction tests whether a channel contains a message by the current time which is held in the T register. The test is performed on the entry in the agent's I register corresponding to the channel being tested. Assume that this entry is (p,l) and that the current value of the agent's time register is T. Let the result of the READY instruction be denoted by r.

```
r = If IsNil(l) Or (l = error)                              (7.6)
    Then false
    Else If timestamp > T
         Then false
         Else true
    where
    timestamp = Head(Head(l)) ;
```

If an out of date message has been received on p then any subsequent **Ready** test is false. If l is Nil the **Ready** test is also false since any further messages on p must have timestamps greater than T or they will cause a timeout error. This allows the ability of the **Ready** test to timeout messages to be implemented. Finally, if there is a message in l then the result of **Ready** can be determined from its timestamp.

7.3.4 Conclusion

In this section we have outlined a possible implementation for **Ruth**. The purpose was to show that such an implementation presents no major difficulties and the two-level approach chosen was dictated by simplicity and ease of explanation. A more sophisticated implementation using some of the techniques detailed in [Peyton Jones 87] would almost certainly prove more efficient. An efficient implementation of **Ruth** is not, however, the purpose of this work.

7.4 Theoretical Issues

One of the major reasons for giving a formal semantics for a programming language is to facilitate formal reasoning and correctness preserving transformation. In this section we examine just what kind of reasoning and transformation is possible in a real-time context.

7.4.1 Reasoning with real-time programs

Consider the following **Ruth** function definition.

```
f = lambda (n). ConsCh ('Any', n + 10, f(n + 10))          (7.7)
```

In the absence of timeouts the expression `f(0)` would produce the channel

```
<10, ['Any', <20, ['Any', <...>]>]>                        (7.8)
```

but if a timeout occurs at any point the remainder of the channel will be $<\infty, \perp>$. Whether or not a timeout occurs depends on the values held in the semantic clock about which we have no information. A similar effect is caused by the `filter` function at process boundaries.

Since the message timestamps in a channel denote exactly when the messages will arrive at their destinations we have the situation that *in the absence of timeouts* it is possible to prove not just *what* data values are produced by a **Ruth** program but also *when* those values will be produced. If a timeout does occur then no further messages will be produced.

It might appear that we are able to prove a form of partial correctness : the right messages will be produced at the right time or no messages will be produced at all. However this is not the whole story. In a real-time system a failure to produce results conveys (almost) as much information as the results themselves and it is to detect such situations that **Ruth** has the **Ready** test.

Consider two possible evaluations of `f(0)` : (i) in which the first message is timed out and (ii) in which the first message is not timed out.

(i) **Ready** `(f(0),15) = true` if no timeout occurs (7.9)
(ii) **Ready** `(f(0),15) = false` if timeout occurs

Since the occurrence of a timeout depends upon the values in the semantic clock there is no way in which the result of a **Ready** test can be predicted from the text of a **Ruth** program.

Were the **Ready** test to be removed from **Ruth** we would at least have the ability to prove partial correctness : that if any answers are produced they will be the "correct" ones. It would, however, be impractical to omit the **Ready** test from **Ruth** for reasons mentioned above.

If it were possible to predict when timeouts would occur we would have a total solution to the problem : the results of a **Ruth** program could be predicted from the text of the program. The major advantage of clock-driven timing is also its major disadvantage : all the information about time is abstracted in to the semantic clocks leaving us no way of reasoning about the occurrence of timeouts.

7.4.2 Correctness preserving transformation with real-time programs

The basis for formal transformation of program is the notion of referential transparency : equivalent expressions can be interchanged at will. Consider the following two expressions

(i) `4` (7.10)
(ii) `2 + 2`

Suppose the result of (i) is $<t_1, 4>$ for some time t_1, and the result of (ii) is $<t_2, 4>$ for some time t_2. It can be seen from the semantics that t_1 and t_2 will be different : t_1 will be the first time value in the semantic clock used to evaluate the expression whereas t_2 will be read from the 0th sub-clock of that clock and must therefore be bigger than t_1.

Under a non real-time semantics (i) and (ii) would be considered equivalent since the data value part of their result is all that is considered : the classical notion of referential transparency abstracts from the real-time aspect. Paraphrasing [Stoy 77] we can better express classical referential transparency as follows.

> The only thing that matters about an expression are its **data** value **and its semantic timestamp** value, and any expression can be replaced by any other with equal **data** value.

The addition of the consideration of time to the definition of classical referential transparency has made explicit what was previously implicit : that for most computational purposes two expressions are equivalent if they compute the same data value, regardless of when they compute it. We shall refer to this interpretation of classical referential transparency as **data referential transparency** (DRT).

A more "hard-line" interpretation of referential transparency is what we shall call **real-time referential transparency** (RRT) which is defined as follows.

> The only thing that matters about an expression are its **data** value **and its semantic timestamp** value, and any can expression can be replaced by any other with equal **data** value **and semantic timestamp** value.

Under RRT only totally identical expressions whose evaluations are carried out at the same time can be considered to be equivalent. Consequently RRT is not a useful basis for program transformation since one of the major reasons for transforming programs is to make them execute faster : to change the values of the semantic timestamps. DRT, on the other hand, is the basis for many transformation systems on non-real time languages; for a survey see [Partsch & Steinbruggen 83].

Some transformation systems (e.g. [Sherlis 80]) guarantee the preservation of total correctness : if evaluation of expression E terminates with the result v then evaluation of the transformed version of E, E^T, will also terminate with the result v. Other systems (e.g. [Darlington 82]) guarantee to preserve only partial correctness : if evaluation of expression E terminates with the result v then *if* evaluation of the transformed version of E, E^T, also terminates it will do so with the result v.

In a real-time semantics total correctness preservation is expressed as **data total correctness** (DTC) preservation, and partial correctness preservation is expressed as **data partial correctness** (DPC) preservation.

Definition : Data total correctness preservation. (7.11)
Let E be a **Ruth** Expression and let E^T be E transformed by some transformation T.
If $\mathcal{E}_V [\![E]\!] \rho\ c = \langle t,v\rangle$ and $\mathcal{E}_V [\![E^T]\!] \rho\ c = \langle t^T,v^T\rangle$
Then the transformation T preserves DTC iff $v = v^T$

Definition : Data partial correctness preservation.
Let E be a **Ruth** Expression and let E^T be E transformed by some transformation T.
If $\mathcal{E}_V [\![E]\!] \rho\ c = \langle t,v\rangle$ and $\mathcal{E}_V [\![E^T]\!] \rho\ c = \langle t^T,v^T\rangle$
Then the transformation T preserves DPC iff $v \sqsubseteq v^T$ or $v^T \sqsubseteq v$

The question is, can DTC or DPC preservation be used as a basis for program transformation in real-time systems? Let ch be an output channel from a **Ruth** process P and let ch^T the corresponding output channel from the transformed process P^T. If the transformation from P to P^T preserves DPC then either ch is a prefix of ch^T, or ch^T is a prefix of ch. If the transformation from P to P^T preserves DTC then ch is identical to ch^T.

The problem with DPC preservation is the same problem as was encountered in the last section on formal reasoning : **Ready**'s ability to detect the timing out of a message due to a non-terminating, or simply too slow, computation. By transforming a process or a channel construction it is possible to introduce, or to remove, timeouts and thus potentially change the data value part of the result of a **Ready** test.

Once again, if the **Ready** test is removed from **Ruth** then DPC preserving transformation becomes possible. Indeed, a transformation technique which preserves classical partial correctness would also preserve DPC when used with **Ruth** programs. However removing the **Ready** test would remove most of **Ruth**'s power as a real-time language. Furthermore DPC preservation is not acceptable in real-time systems because it allows for timeouts to be introduced by transformations. In non real-time situations a tardy computation can be aborted and another attempt made. Real-time systems do not allow second attempts.

The same problem exists for DTC as for DPC : the introduction and/or removal of timeouts must be avoided. The situation is more serious than for DPC. Even if the `Ready` test were removed from **Ruth** DTC preserving transformation is difficult since a transformation which preserves classical total correctness may not preserve DTC. Although the transformed expression will terminate provided the original terminated it may take a different amount of time to do so. Thus timeouts may be introduced or removed and thus DTC will not be preserved.

Simply being able to predict the occurrence of timeouts would solve these problems since transformations which introduced or removed timeouts could be identified and thus DTC preservation could be guaranteed. Because channel message timestamps are part of the data that would be preserved, we would thus gain the ability to transform programs and still preserve their real-time behaviour.

7.4.3 Conclusions

The biggest problem with formal reasoning and transformation of **Ruth** programs is the combination of clock-driven semantics and channel message timeouts. Clock-driven semantics allows abstraction away from low-level detail but this abstraction makes it impossible to reason about the values of semantic timestamps save in the most general terms. In particular, it is impossible to predict when timeouts will occur at channel construction or process boundaries.

Ruth's time expressibility is based upon user-defined timestamps to identify when events will, or did, occur, and upon a determinate `Ready` test for detecting the occurrence or non-occurrence of an event. Thus, in the absence of timeouts, the real-time behaviour of a **Ruth** program is totally defined by, and can therefore be proved from, the program's text. Also, DTC preserving transformation is sufficient to preserve the real-time behaviour of a **Ruth** program.

Most other real-time languages use implicit *don't care/don't know* timestamping to identify when events should/did occur, and implicit time determinate operators to detect the non-occurrence of events. By *don't care* timestamping we mean that an event will occur *as soon as possible*, and by *don't know* timestamping we mean that the system does not know when an event occurred save that it occurred before the current time. The real-time behaviour of a program written in such a language is totally defined by the speed at which it is executed. The way this would be modelled semantically depends upon whether a clock- or delay-driven semantics is used. Using a clock-driven semantics timestamps would depend on the (unknown) values in the semantic clock. Using a delay-driven semantics timestamps would depend upon the δ values specified for the operations required to compute data values.

If a delay-driven semantics is specified then it is possible to prove the real-time behaviour of a language using don't care/don't know timestamping, though performing such a proof is probably impractical for all but the most trivial programs. If a clock-driven semantics is given proof of real-time behaviour is impossible. This is slightly worse than with **Ruth** for which, even using a clock-driven semantics, it is possible to prove a program's real-time behaviour assuming the absence of timeouts.

For either a delay-driven or clock-driven semantics DTC preserving transformation will not, in general, preserve the real-time behaviour of a language using don't care/don't know timestamping. In this context DTC preserving transformation will only guarantee to preserve the data values computed, not the times at which they are computed. This is much worse than with **Ruth** for which DTC preserving transformation will preserve real-time behaviour.

Thus, **Ruth** offers certain advantages for formal reasoning and transformation over a traditional real-time language with time expressibility based on don't care/don't know timestamping. To a large extent, however, these advantages require that timeouts be detectable from the text of a **Ruth** program. In the absence of timeouts the real-time behaviour of a **Ruth** program is totally determined by its text, and this behaviour will not be changed by a DTC preserving transformation since such a transformation will neither introduce, nor remove, timeouts. The remaining problem is then to detect timeouts within **Ruth** programs. If a **Ruth** program can be proved timeout free then its real-time behaviour is guaranteed. If both the source and result expressions of a classical total correctness preserving transformation can be proved timeout free then the transformation has preserved DTC.

A possible approach to this problem is simply to assume that timeouts do not occur. The real-time behaviour specified by programs can be proven modulo this assumption. Classical total correctness preserving transformations can be applied to programs since, in the absence of timeouts, they will preserve DTC, and thus real-time behaviour. The transformations used will be *go-faster* transformations : transformations which produce an expression which computes the same result as the original expression but with a lower semantic timestamp value (i.e. the transformed expression produces a result faster). By applying go-faster transformation the hope is that the potential for the occurrence of timeouts will be removed.

Of course, this is a totally heuristic method since, under a clock-driven semantics, there is no way in which go-faster transformations can be identified. Equally, there is no certainty that the program which results from transformation is timeout free anyway.

The situation can be improved by giving a delay-driven semantics for **Ruth**. To avoid problems caused by the possible variability in duration of language operations the δ-values chosen will be the maximum that the relevant operation can take. Consequently, this approach is called *defensive* delay-driven timing. Any putative implementation of **Ruth** which did not guarantee to exactly conform to these δ-values would not be a valid implementation of the language.

The advantage of a defensive delay-driven semantics is that timeouts can always be predicted and thus DTC preserving, go-faster transformations identified. A disadvantage of a defensive delay-driven timing semantics is the difficulty of choosing the correct δ-values. If these values are too small the language may be difficult to implement; too large and the language may be too slow to be useful. A further disadvantage is that a defensive delay-driven semantics must specify evaluation strategy in some detail. For example, if a lazy evaluation mechanism is used then the exact way in which recipes are updated with their values must be specified in the high level semantics. Also, to prevent δ-values becoming too large an incremental garbage collection mechanism must be used and multi-programming cannot be allowed.

The next section outlines a compromise in which as few as possible of the constructs of a language need be given a defensive delay-driven semantics in order to guarantee real-time behaviour. By restricting the set of constructs chosen to those for which a fixed, and small, δ-value can be guaranteed it is hoped that the problems mentioned above will be minimised.

7.5 A Language Based On Explicit Timeouts

7.5.1 Introduction

As was seen in the **Minesweep** program, real-time software often has the following structure :

```
loop                                                    (7.12)
  Get inputs ;
  Compute outputs ;
  Send outputs ;
endloop
```

that is, an infinite sequence of I/O interactions and computation steps. The **Minesweep** program contained output supervisors to provide the following, defensive, behaviour :

```
start algorithm ;                                       (7.13)
if algorithm produces result within $t_1$ of starting
then ok
else interrupt algorithm and return default value
```

If t_1 time units have passed and the algorithm has not yet produced a result then it is pre-emptively interrupted and a default constant is returned instead.

The underlying concept here is that the programmer expects that each interaction step will take a fixed, and known, amount of time, whereas each computation step may take an indefinite amount of time, depending upon the parameters of the computation and the current state of the machine. Consequently, to ensure that time deadlines are met a means of preempting a computation step and taking recovery action is required.

In this section we introduce a programming language which allows real-time programs to be written in the style outlined above. The language is called **STRuth**, which stands for "Sequential, Timeout **Ruth**" since it provides explicitly sequential constructs and a facility for preemptive timeouts, and is an attempt to put into practice the lessons learned from **Ruth**.

7.5.2 The language **STRuth**

In the same way as real-time programs can be divided into interactions and computations **STRuth** is divided into a behavioural and a computational language. As noted above even a functional program tends to interact with its environment in a sequential manner and it therefore seems sensible to provide sequential primitives to allow this. The notation we shall use here, whilst appearing very similar to traditional imperative language notation, is in fact no more than a syntactic transformation of purely functional combinators such as those used in [Thompson 86] and [Jones & Sinclair 89].

The most important feature of the behavioural language is that each of its constructs takes a fixed amount of time to execute.Thus a defensive delay-driven timing semantics can be given for the behavioural language and formal reasoning and transformation is possible with it.

All interaction with the environment, and the times at which it occurs, is expressed in the behavioural language. Thus the computational language need have no notion of time expressibility; in particular, the **Ready** test and channel primitives are not required. A traditional functional language, such as Haskell [Hudak et. al. 89], is thus perfectly adequate for the task. Operations in the computational language are not expected to take a fixed amount of time to execute; instead the behavioural language provides facilities for tardy computations to be timed out. Since fixed δ-values need not be specified, a clock-driven timing semantics is appropriate for the computational language.

The simple, and somewhat contrived, program overleaf repeatedly reads an integer from the input channel `in1`, a boolean from the input channel `in2`, and the current time from the pre-defined input channel `time`. If the input boolean is `true` then the integer and the time are output on output channel `out1`, otherwise they are output on output channel `out2`.

```
chan input in1, in2 ; output out1, out2 ;                        (7.14)
var x,y : integer ; b : bool ;

behaviour
  loop true do
    in1 ? x ; in2 ? b ;  time ? y ;

    if b then
      out1 ! y ; out1 ! x
    else
      out2 ! y ; out2 ! x
    endif

  endloop
endprog
```

The example contains all of the behavioural constructs allowed in **STRuth** apart from assignment : identifier reference, loop, if-then-else, input and output. Since a fixed duration is guaranteed for identifier reference, input, output and assignment, a fixed duration can be guaranteed for any loop or if-then-else, and thus for any complete **STRuth** program.

A **STRuth** assignment is of the form

```
v := "computational language expression" ;                       (7.15)
```

Obviously, once the computational language expression has produced a value it takes a fixed, and specifiable, time to bind that value to v. However, there is no way of determining how long a computational language expression may take to evaluate.To ensure that deadlines are not missed the programmer is allowed to specify the completion time for any assignment. If the assignment is not completed by that time then an interrupt occurs and recovery action can be taken.

```
x := E timeout 800 ;                                             (7.16)
```

An attempt is made to evaluate the computational language expression E. If this evaluation is not completed and its result successfully bound to x within 800 time units of the start of the assignment then an interrupt occurs and execution of the assignment terminates with the value bound to x unchanged (i.e. the same as before the assignment started).

If **timeout** 800 were omitted then as much time as required would be taken for E, and the result would be bound to x with no possibility of timeout. In this case there is no way of determining when the assignment takes place.

A similar approach can be taken to the problem of guaranteeing that input/output interactions terminate within a fixed time.

```
in ? x timeout 800 ;                                    (7.17)
```

If a message is not available on channel in within 800 time units of the start of execution of the input then an interrupt occurs and execution of the input terminates with the value bound to x unchanged.

7.5.3. Formal semantics

The meaning of a **STRuth** program is defined via the evaluation function $\mathcal{E}_{\mathrm{P}}$ which has the following signature.

$$\mathcal{E}_{\mathrm{P}} : \mathrm{PROG} \rightarrow \mathbf{ENV} \rightarrow \mathbf{NUM} \rightarrow \mathbb{CLK} \rightarrow \mathbf{ENV} \qquad (7.18)$$

Here PROG is the syntactic domain of **STRuth** programs. A **STRuth** program is a mapping from an initial environment, the current time represented as a positive integer and a clock, to a final environment. The clock argument is not used by $\mathcal{E}_{\mathrm{P}}$ but is passed to the meaning function $\mathcal{E}_{\mathrm{V}}$ which defines the semantics of a computational language expression.

$$\mathcal{E}_{\mathrm{V}} : \mathrm{Exp} \rightarrow \mathbf{ENV} \rightarrow \mathbb{CLK} \rightarrow \mathbb{VAL} \qquad (7.19)$$

Exp is the syntactic domain of computational language expressions. A computational language expression is a mapping from an environment and a clock to an expressible value.

The semantics of a simple assignment are as follows

$$\begin{array}{l} \mathcal{E}_P [\![\; x := E_1 \; ; \; P \;]\!] \; \rho \; t \; c \qquad\qquad\qquad\qquad\qquad\qquad (7.20) \\ \quad = \langle\!\langle \; \langle t_1, v_1 \rangle \; : \; \mathcal{E}_V [\![\; E_1 \;]\!] \; \rho \; \mathrm{clockafter}(c_0, t) \\ \qquad\qquad \to \mathcal{E}_P [\![\; P \;]\!] \; (\rho \; \& \; [x \to \langle t_1, v_1 \rangle]) \; (t_1 + \delta_{assign}) \; c_1 \\ \qquad \rangle\!\rangle \end{array}$$

P, the remainder of the program, is executed with an environment in which x is bound to the value obtained by evaluating the computational expression E_1. Evaluation of E_1 does not begin until after t; there is no guarantee of when evaluation of E_1 terminates. δ_{assign} is the time taken after E_1 has been evaluated to bind its result to x; execution of P does not begin until $t_1 + \delta_{assign}$.

To give the semantics of an assignment which may be timed out requires the use of $\mathbf{Match} \ldots \mathbf{with} \ldots$ notation rather than the more secure $\langle\!\langle \; \ldots \; \rangle\!\rangle$ since timeout behaviour must be specified (c.f. the definition of **Ready** in Chapter 5).

$$\begin{array}{l} \mathcal{E}_P [\![\; x := E_1 \; \mathbf{timeout} \; 800 \; ; \; P \;]\!] \; \rho \; t \; c \qquad\qquad\qquad\qquad (7.21) \\ \quad = (\mathbf{Match} \; \langle t_1, v_1 \rangle \; \mathbf{with} \\ \qquad\quad \langle t_1, \bot \rangle \to \; ((t_1 + \delta_{assign}) \leq (t + 800) \\ \qquad\qquad\qquad\qquad \to \langle t_1, \bot \rangle, \\ \qquad\qquad\qquad\qquad\quad \mathcal{E}_P [\![\; P \;]\!] \; \rho \; (t + 800) \; c_1 \\ \qquad\qquad\qquad) \\ \qquad\quad \langle t_1, v_1 \rangle \to \; ((t_1 + \delta_{assign}) \leq (t + 800) \\ \qquad\qquad\qquad\qquad \to \mathcal{E}_P [\![\; P \;]\!] \; (\rho \; \& \; [x \to \langle t_1, v_1 \rangle]) \; (t + 800) \; c_1, \\ \qquad\qquad\qquad\qquad\quad \mathcal{E}_P [\![\; P \;]\!] \; \rho \; (t + 800) \; c_1 \\ \qquad\qquad\qquad) \\ \qquad) \\ \qquad \mathbf{where} \\ \qquad \langle t_1, v_1 \rangle = \mathcal{E}_V [\![\; E_1 \;]\!] \; \rho \; \mathrm{clockafter}(c_0, t) \end{array}$$

Note that here we have been a little simplistic in that we have assumed that the timeout time is the fixed constant, 800, rather than a value computed during execution of the program. The general principle holds good however : if the value of E_1 can be computed in time for its result to be bound to x by absolute time t + 800 then this is done. Otherwise an interrupt will occur and the binding for x will be left unchanged. In either case execution of P, the rest of the program, will not begin until t + 800. **STRuth** takes a *defensive* view : since there is no way to predict when evaluation of E_1 will terminate, all that can be guaranteed is that the assignment will terminate by the interrupt time, t + 800. Even if evaluation of E_1 terminates quickly enough for the assignment to be completed before t + 800 the program must still wait until t + 800 before starting P. This may seem excessively severe but is the only way to guarantee the real-time behaviour of a timed out assignment.

The semantics for an input are similar.

```
$\mathcal{E}_P$ [[ in ? x timeout 800 ; P ]] $\rho$ t c                         (7.22)
  = (Match <$t_1$ + $\delta_{Id}$, $v_1$> with
      <t',$\perp$> $\rightarrow$  ((t' + $\delta_{assign}$) $\leq$ (t + 800)
                      $\rightarrow$ <t',$\perp$>,
                        $\mathcal{E}_P$ [[ P ]] $\rho$ (t + 800) $c_1$
                 )
      < t',[a,rest]>
               $\rightarrow$  ((t' + $\delta_{assign}$) $\leq$ (t + 800)
                      $\rightarrow$ $\mathcal{E}_P$ [[ P ]] ($\rho$ & [x $\rightarrow$ <t',a>]) (t + 800) $c_1$,
                        $\mathcal{E}_P$ [[ P ]] $\rho$ (t + 800) $c_1$
                 )
    )
    where
    <$t_1$,$v_1$> = lookup(in,$\rho$)
```

There is no identification timestamp associated with the channel bound to the identifier in. Identification timestamps were required for **Ruth** channels because a channel could be the result of an arbitrarily lengthy computation. In **STRuth** the only way to reference a channel is via an identifier reference as above. Identifier reference has a fixed overhead (δ_{Id} above) so no identification timestamp is required.

7.5.4 Conclusion

The principles behind **STRuth** are that to give a delay-driven semantics for a complete real-time programming language is difficult, and that such a semantics is probably impractical to work with at present. Instead, a clock-driven semantics is used for most of **STRuth** and the use of defensive delay-driven semantics is restricted to a small subset. Programs written in **STRuth** guarantee either to interact with their environments at the correct times, or with the correct data values. In the absence of a complete delay-driven semantics it is impossible for a language to guarantee both. **STRuth** accepts that fact and provides programmers with the ability to decide which aspect of correctness is the more important. If correct data values are more important, then a combination of inputs, simple assignments and outputs suffices. If deadlines must be met, then **STRuth** provides an explicit timeout construct to prevent computations from overrunning. An alternative may be substituted for the timed out computation and the deadline still met.

The work presented here is of a very preliminary nature. Nonetheless, it seems likely that this approach will prove valuable in the production of reliable real-time software, and represents the most promising route for further research. For further information about **STRuth** the reader is referred to [Harrison & Nelson 89].

7.6 Conclusion

This thesis has explored the application of functional programming techniques to real-time programming via the design of the real-time functional programming language **Ruth**. Many existing real-time languages have poor facilities for time expressibility and programmers often rely upon their intuitive knowledge of how a von Neumann processor works to ensure that things happen at the right times. Moreover, implicit time determinate operators are usually used to allow for non-blocking tests for input. Neither of these approaches is possible within a functional language and we were forced to evolve new methods. Time expressibility was provided via timestamps and real-time clock streams, and non-blocking test for input via the `Ready` test which is a determinate test on user-defined message timestamps.

The major part of this work is the semantic definition of **Ruth** using herring-bone domains and clock-driven timing. An exploration of a framework within which the real-time characteristics of computer programming languages could be expressed was not the original intention but, in retrospect, has probably proven the most interesting result of this work, and probably the most useful.

Ruth is by no means a perfect real-time programming language and, as we have seen, certain difficulties exist in formal reasoning and transformation of **Ruth** programs. These difficulties are a direct consequence of working within the real-time domain : since when values are computed is a part of the result of a real-time program is comes as no surprise that this information must be supplied by giving a defensive delay-driven semantics to the language. Unfortunately giving a defensive delay-driven semantics for a language is a difficult task. The explicit message timestamps used in **Ruth** programs limit these problems but do not totally solve them. **STRuth** extends the explicit specification of timing requirements in programs to a limited set of language operations. It is hoped that a compromise can be reached in which the real-time behaviour of programs can be guaranteed without the need to specify a defensive delay-drive semantics for all language constructs.

Appendix 1 : Formal Presentation Of The Semantic Domains

A1.1 The Foundations

In this section we define the primitive domains and orderings, domain constructors the orderings they impose, and constructors and selectors for the constructed domains.

Firstly the primitive domains

Definition : Primitive Domains (A1.1)

INT	Integers : $\{\perp, \ldots, -2, -1, 0, 1, 2, \ldots\}$
NUM	Integers ≥ 0 : $\{\perp, 0, 1, 2, \ldots\}$
BOOL	Booleans : $\{\perp, \mathtt{true}, \mathtt{false}\}$
STRING	Symbolic strings : $\{\perp, \mathtt{a}, \mathtt{ab}, \mathtt{abc}, \ldots\}$
NIL	End of list marker : $\{\perp, \mathtt{Nil}\}$
UNDEF	Identifier not defined indicator : $\{\perp, \mathtt{Undef}\}$
NUM	Set of integers ≥ 0 : $\{0, 1, 2, \ldots\}$

All primitive domains except NUM have the following ordering

$$x \sqsubseteq y \Leftrightarrow (x = \perp) \vee (x = y)$$

and thus are pointed flat cpos. NUM is a set with no ⊥ element and thus has the ordering

$$x \sqsubseteq y \Leftrightarrow x = y$$

Before proceeding we shall define the syntax of the boolean choice primitive to be used.

Definition : Choice Primitive (A1.2)

For any domain D

$$(\_ \rightarrow \_,\_) \; : \; (\mathbf{BOOL} \times \mathbf{D} \times \mathbf{D}) \rightarrow \mathbf{D}$$

such that

$$(\perp \rightarrow d_1, d_2) = \perp$$

$$(\mathtt{true} \rightarrow d_1, d_2) = d_1$$

$$(\mathtt{false} \rightarrow d_1, d_2) = d_2$$

The definitions of the domain constructors to be used, he orderings the constructors impose on the constructed domains, and constructors and selectors for elements of the constructed domains are given below.

Definition : Lifted Domain (A1.3)
For any domain $\mathbf{A}$

Elements : $\mathbf{A}_\perp = \{\mathtt{lift}(a) \mid a \in \mathbf{A}\} \cup \{\perp\}$

Ordering :

$$
\begin{aligned}
x \sqsubseteq_\perp y \Leftrightarrow\ & (x = \perp) \vee \\
& (x = \mathtt{lift}(a_x) \wedge y = \mathtt{lift}(a_y) \wedge a_x \sqsubseteq_A a_y)
\end{aligned}
$$

Constructor :

$$
\begin{aligned}
\perp &: \quad \rightarrow \mathbf{A}_\perp \\
\mathtt{lift} &: \mathbf{A} \rightarrow \mathbf{A}_\perp
\end{aligned}
$$

Selector : By pattern matching on $\mathtt{lift}(a)$ elements

Definition : Sum Domain (A1.4)
For any two domains $\mathbf{A}$ and $\mathbf{B}$

Elements : $\mathbf{A} + \mathbf{B} = \{\mathtt{in}\mathbf{A}(a) \mid a \in \mathbf{A}\} \cup \{\mathtt{in}\mathbf{B}(b) \mid b \in \mathbf{B}\} \cup \{\perp\}$

Ordering :

$$
\begin{aligned}
x \sqsubseteq_{\mathbf{A+B}} y \Leftrightarrow\ & (x = \perp) \vee \\
& (x{=}\mathtt{in}\mathbf{A}(a_x) \wedge y{=}\mathtt{in}\mathbf{A}(a_y) \wedge a_x \sqsubseteq_{\mathbf{A}} a_y) \vee \\
& (x{=}\mathtt{in}\mathbf{B}(b_x) \wedge y{=}\mathtt{in}\mathbf{B}(b_y) \wedge b_x \sqsubseteq_{\mathbf{B}} b_y)
\end{aligned}
$$

Constructors :

$$
\begin{aligned}
\perp &: \quad \rightarrow \mathbf{A+B} \\
\mathtt{in}\mathbf{A} &: \mathbf{A} \rightarrow \mathbf{A+B} \\
\mathtt{in}\mathbf{B} &: \mathbf{B} \rightarrow \mathbf{A+B}
\end{aligned}
$$

Selectors :
such that

$$
\begin{aligned}
& (\mathtt{Cases}\ x\ \mathtt{of}\ \mathtt{is}\mathbf{A}(a) \rightarrow e_1,\ \mathtt{is}\mathbf{B}(b) \rightarrow e_2) \\
& (\mathtt{Cases}\ \mathtt{in}\mathbf{A}(a)\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \rightarrow e_1,\ \mathtt{is}\mathbf{B}(y) \rightarrow e_2) \\
& \quad = (\lambda(x).e_1)\ (a) \\
& (\mathtt{Cases}\ \mathtt{in}\mathbf{B}(b)\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \rightarrow e_1,\ \mathtt{is}\mathbf{B}(y) \rightarrow e_2) \\
& \quad = (\lambda(y).e_2)\ (b) \\
& (\mathtt{Cases}\ \perp\ \mathtt{of}\ \mathtt{is}\mathbf{A}(x) \rightarrow e_1,\ \mathtt{is}\mathbf{B}(y) \rightarrow e_2) \\
& \quad = \perp
\end{aligned}
$$

and also $(\text{Cases } x \text{ of is}\mathbf{A}(a) \rightarrow e_1, \text{ else} \rightarrow e_2)$

such that $(\text{Cases in}\mathbf{A}(a) \text{ of is}\mathbf{A}(x) \rightarrow e_1, \text{ else} \rightarrow e_2)$

$= (\lambda(x).e_1)\ (a)$

$(\text{Cases in}\mathbf{B}(b) \text{ of is}\mathbf{A}(x) \rightarrow e_1, \text{ else} \rightarrow e_2)$

$= e_2$

$(\text{Cases } \bot \text{ of is}\mathbf{A}(x) \rightarrow e_1, \text{ is}\mathbf{B}(y) \rightarrow e_2)$

$= \bot$

Definition : Product Domain (A1.5)

For any two domains **A** and **B**

Elements : $\mathbf{A} \times \mathbf{B} = \{[a,b] \mid a \in \mathbf{A},\ b \in \mathbf{B}\}$

Ordering : $[a,b] \sqsubseteq_{A \times B} [a',b'] \Leftrightarrow [a \sqsubseteq_A a'] \wedge [b \sqsubseteq_B b']$

Constructor : $[\_,\_] : \mathbf{A} \times \mathbf{B} \rightarrow \mathbf{A} \times \mathbf{B}$

Selectors : By pattern matching on [a,b] elements

Definition : Coalesced Sum Domain (A1.6)

For any two domains **A** and **B**

Elements : $\mathbf{A} \oplus \mathbf{B} = \{\text{in}\mathbf{A}(a) \mid a \in \mathbf{A},\ a \neq \bot\} \cup$

$\{\text{in}\mathbf{B}(b) \mid b \in \mathbf{B},\ b \neq \bot\} \cup$

$\{\bot\}$

Ordering : $x \sqsubseteq_{A \oplus B} y \Leftrightarrow (x = \bot) \vee$

$(x=\text{in}\mathbf{A}(a_x) \wedge y=\text{in}\mathbf{A}(a_y) \wedge a_x \sqsubseteq_A a_y) \vee$

$(x=\text{in}\mathbf{B}(b_x) \wedge y=\text{in}\mathbf{B}(b_y) \wedge b_x \sqsubseteq_B b_y)$

Constructors : $\bot \quad : \quad \rightarrow \mathbf{A} \oplus \mathbf{B}$

$\text{in}\mathbf{A} : \mathbf{A} \rightarrow \mathbf{A} \oplus \mathbf{B}$ where $\text{in}\mathbf{A}(\bot_A) = \bot$

$\text{in}\mathbf{B} : \mathbf{B} \rightarrow \mathbf{A} \oplus \mathbf{B}$ where $\text{in}\mathbf{B}(\bot_B) = \bot$

Selectors : $(\texttt{Cases } x \texttt{ of is}\mathbf{A}(a) \rightarrow e_1,\ \texttt{is}\mathbf{B}(b) \rightarrow e_2\ )$

such that $(\texttt{Cases in}\mathbf{A}(a) \texttt{ of is}\mathbf{A}(x) \rightarrow e_1,\ \texttt{is}\mathbf{B}(y) \rightarrow e_2)$

$= (\lambda(x).e_1)\ (a)$

$(\texttt{Cases in}\mathbf{B}(b) \texttt{ of is}\mathbf{A}(x) \rightarrow e_1,\ \texttt{is}\mathbf{B}(y) \rightarrow e_2)$

$= (\lambda(y).e_2)\ (b)$

$(\texttt{Cases } \perp \texttt{ of is}\mathbf{A}(x) \rightarrow e_1,\ \texttt{is}\mathbf{B}(y) \rightarrow e_2)$

$= \perp$

and also $(\texttt{Cases } x \texttt{ of is}\mathbf{A}(a) \rightarrow e_1,\ \texttt{else} \rightarrow e_2)$

such that $(\texttt{Cases in}\mathbf{A}(a) \texttt{ of is}\mathbf{A}(x) \rightarrow e_1,\ \texttt{else} \rightarrow e_2)$

$= (\lambda(x).e_1)\ (a)$

$(\texttt{Cases in}\mathbf{B}(b) \texttt{ of is}\mathbf{A}(x) \rightarrow e_1,\ \texttt{else} \rightarrow e_2)$

$= e_2$

$(\texttt{Cases } \perp \texttt{ of is}\mathbf{A}(x) \rightarrow e_1,\ \texttt{else} \rightarrow e_2)$

$= \perp$

Definition : Coalesced Product Domain (A1.7)

For any two domains **A** and **B**

Elements : $\mathbf{A} \otimes \mathbf{B} = \{[a,b] \mid a \in \mathbf{A},\ a \neq \perp,\ b \in \mathbf{B},\ b \neq \perp\} \cup \{\perp\}$

Ordering : $x \sqsubseteq_{\mathbf{A}\otimes\mathbf{B}} y \Leftrightarrow (x = \perp) \vee$

$(x=[a,b] \wedge y=[a',b'] \wedge$

$a \sqsubseteq_{\mathbf{A}} a' \wedge b \sqsubseteq_{\mathbf{B}} b')$

Constructor : $\perp \quad : \quad \rightarrow \mathbf{A} \otimes \mathbf{B}$

$[\_,\_] \ : \mathbf{A} \times \mathbf{B} \rightarrow \mathbf{A} \otimes \mathbf{B}$ where $[\perp_{\mathbf{A}},b] = \perp$

where $[a,\perp_{\mathbf{B}}] = \perp$

Selectors : By pattern matching on $[a,b]$ elements

Definition : Function Space (A1.8)

For any two domains **A** and **B**

Elements : $\mathbf{A} \rightarrow \mathbf{B} = \{f \mid a \in \mathbf{A},\ f(a) \in \mathbf{B},\ f$ continuous$\}$

Ordering : $f \sqsubseteq_{\mathbf{A}\rightarrow\mathbf{B}} g \Leftrightarrow \forall a \in \mathbf{A},\ f(a) \sqsubseteq_{\mathbf{B}} g(a)$

Constructor : $\lambda x.e \in \mathbf{A} \rightarrow \mathbf{B}$

Selector : $\_(\_) \ : (\mathbf{A} \rightarrow \mathbf{B}) \times \mathbf{A} \rightarrow \mathbf{B}$

A1.2 Herring-Bone Domains

This section concerns the construction of herring-bone domains. Firstly, the isomorphism between $\mathbb{BOOL}$, the herring-bone domain of booleans and $\mathbb{LBOOL}$, the domain constructed using domain lifting is proven. The rule for constructing a herring-bone domain from any arbitrary domain is then given.

The herring-bone domain of booleans is defined as follows :-

Definition : Herring-bone domain of booleans (A1.9)
For the domain of booleans, BOOL, the corresponding herring-bone domain is $\mathbb{BOOL}$.

Elements : $\mathbb{BOOL} = \{<t,b> \mid t \in \text{NUM},\ b \in \textbf{BOOL}\} \cup \{<\infty,\perp>\}$

Ordering : $\forall\ t_1,t_2 \in \text{NUM},\ b_1,b_2 \in \textbf{BOOL}\ :$

$$<t_1,b_1> \sqsubseteq <t_2,b_1> \Leftrightarrow (t_1 = t_2 \wedge b_1 \sqsubseteq_{\textbf{BOOL}} b_2) \vee (t_1 \le t_2 \wedge b_1 = \perp)$$

and $<t_1,\perp> \sqsubseteq <\infty,\perp>$

Constructors :

$<\_,\ \_>\ :\ \text{NUM} \times \textbf{BOOL} \rightarrow \mathbb{BOOL}$

$<\infty,\ \perp>\ :\ \rightarrow \mathbb{BOOL}$

Selectors :

(Match <t,v> with <t',⊥> → e_1, <t',b'> → e_2)

such that

(Match <t,⊥> with <t',⊥> → e_1, <t',b'> → e_2)

= $(\lambda(t').e_1)\ (t)$

(Match <t,b> with <t',⊥> → e_1, <t',b'> → e_2)

= $(\lambda(t',b').e_2)\ (t,b)$ where $b \ne \perp$

Proposition : (A1.10)

The domain $\mathbb{BOOL}$ is isomorphic to the domain $\mathbb{LBOOL}$ defined by
$\mathbb{LBOOL} = \mathbf{BOOL} \oplus \mathbb{LBOOL}_\perp$ as follows

$$\forall\ t \in \mathrm{NUM},\ b \in \mathbf{BOOL},\ b \neq \perp :$$

$$<t,b> = (\lambda(x).\mathrm{in}\mathbb{LBOOL}_\perp(\mathrm{lift}(x)))^t\ (\mathrm{in}\mathbf{BOOL}\ (b))$$

$$<t,\perp> = (\lambda(x).\mathrm{in}\mathbb{LBOOL}_\perp(\mathrm{lift}(x)))^t\ (\perp)$$

$$<\infty,\perp> = \bigsqcup\{\ (\lambda(x).\ \mathrm{in}\mathbb{LBOOL}_\perp(\mathrm{lift}(x)))^t\ (\perp)\ \mid\ t \geq 0\}$$

where, for any function f

$$f^0(x) = x$$

$$f^{t+1}(x) = f(f^t(x))$$

The proof is in two parts. Firstly, fixed-point induction is used to prove the isomorphism for the finite elements of $\mathbb{BOOL}$ and $\mathbb{LBOOL}$. The second part of the proof concerns the infinite elements of the two domains. The only infinite element of $\mathbb{LBOOL}$ is the least upper bound of the chain of spine elements of $\mathbb{LBOOL}$,

$$\bigsqcup\{\ (\lambda(x).\ \mathrm{in}\mathbb{LBOOL}_\perp(\mathrm{lift}(x)))^t\ (\perp)\ \mid\ t \geq 0\}\ \mathbb{BOOL},\ <\infty,\perp>,$$

It is proved that the only infinite element of $\mathbb{BOOL}$, $<\infty,\perp>$, is the least upper bound of the chain of spine elements of $\mathbb{BOOL}$,

$$\bigsqcup\{\ <t,\perp>\ \mid\ t \in \mathrm{NUM}\}.$$

This completes the proof of isomorphism.

We first define two functions, `enlift` and `delift` as follows.

Definition : Projection functions `enlift` and `delift`. (A1.11)

```
enlift : 𝔹𝕆𝕆𝕃 → 𝕃𝔹𝕆𝕆𝕃
enlift
  = λ(b) .
      (Match b with
          <t,⊥>
            → (t = 0 →⊥, in𝕃𝔹𝕆𝕆𝕃⊥(lift(enlift(<t-1,⊥>)))
          <t,torf>
            → (t = 0 →inBOOL (torf),
                          in𝕃𝔹𝕆𝕆𝕃⊥(lift(enlift(<t-1,torf>)))
               )
      )
```

```
delift : LBOOL → BOOL
delift
  = λ(lb).
       (lb = ⊥
         → <0,⊥>,
             (Cases lb of
                isBOOL (b)
                  → <0,b>
                isLBOOL⊥(lift(lb'))
                  → <t+1,lb">
                    where
                    <t,lb"> = delift(lb')
             )
       )
```

`enlift` must be proven to be monotonic as it uses $\mathbb{M}\mathbb{a}\mathbb{t}\mathbb{c}\mathbb{h}$.

Proposition : `enlift` is monotonic. (A1.12)

Proof

Base case :

$\forall\ b_1, b_2 \in \mathbf{BOOL}\ \ \texttt{<0,}b_1\texttt{>} \sqsubseteq \texttt{<0,}b_1\texttt{>} \Leftrightarrow b_1 \sqsubseteq b_1$

$\texttt{enlift(<0,}b_1\texttt{>)} = \texttt{in}\mathbf{BOOL}\texttt{(}b_1\texttt{)}$

$\texttt{enlift(<0,}b_2\texttt{>)} = \texttt{in}\mathbf{BOOL}\texttt{(}b_2\texttt{)}$

$\Rightarrow \forall\ b_1, b_2 \in \mathbf{BOOL}\ \texttt{<0,}b_1\texttt{>} \sqsubseteq \texttt{<0,}b_1\texttt{>} \Leftrightarrow$

$\texttt{enlift(<0,}b_1\texttt{>)} \sqsubseteq \texttt{enlift(<0,}b_2\texttt{>)}$

Induction step : Assume that for $\texttt{<}t_1\texttt{,}b_1\texttt{>,<}t_2\texttt{,}b_2\texttt{>} \in \mathbb{BOOL}$,

$\texttt{<}t_1\texttt{,}b_1\texttt{>} \sqsubseteq \texttt{<}t_2\texttt{,}b_2\texttt{>} \Rightarrow \texttt{enlift(<}t_1\texttt{,}b_1\texttt{>)} \sqsubseteq \texttt{enlift(<}t_2\texttt{,}b_2\texttt{>)}$

$\texttt{<}t_1\texttt{+1,}b_1\texttt{>} \sqsubseteq \texttt{<}t_2\texttt{+1,}b_2\texttt{>}$

$\texttt{enlift(<}t_1\texttt{+1,}b_1\texttt{>)} = \texttt{in}\mathbb{LBOOL}_\perp\texttt{(lift(enlift(<}t_1\texttt{,}b_1\texttt{>)))}$

$\texttt{enlift(<}t_2\texttt{+1,}b_2\texttt{>)} = \texttt{in}\mathbb{LBOOL}_\perp\texttt{(lift(enlift(<}t_2\texttt{,}b_2\texttt{>)))}$

$\texttt{enlift(<}t_1\texttt{+1,}b_1\texttt{>)} \sqsubseteq \texttt{enlift(<}t_2\texttt{+1,}b_2\texttt{>)}$ by induction hypothesis

$\texttt{<}t_1\texttt{+1,}b_1\texttt{>} \sqsubseteq \texttt{<}t_2\texttt{+1,}b_2\texttt{>} \Rightarrow \texttt{enlift(<}t_1\texttt{+1,}b_1\texttt{>)} \sqsubseteq \texttt{enlift(<}t_2\texttt{+1,}b_2\texttt{>)}$

$\Rightarrow\ \ \forall\ \texttt{<}t_1\texttt{,}b_1\texttt{>,<}t_2\texttt{,}b_2\texttt{>} \in \mathbb{BOOL}$,

$\texttt{<}t_1\texttt{,}b_1\texttt{>} \sqsubseteq \texttt{<}t_2\texttt{,}b_2\texttt{>} \Rightarrow \texttt{enlift(<}t_1\texttt{,}b_1\texttt{>)} \sqsubseteq \texttt{enlift(<}t_2\texttt{,}b_2\texttt{>)}$

$\Rightarrow$ `enlift` is monotonic

For the finite elements of $\mathbb{BOOL}$ and $\mathbb{LBOOL}$ to be isomorphic we require that

$$\forall\ \mathtt{lb} \in \mathbb{LBOOL},\ \mathtt{lb} \neq \bigsqcup\{\ (\lambda(\mathtt{x}).\ \mathtt{in}\mathbb{LBOOL}_{\perp}(\mathtt{lift(x)}))^{\mathtt{t}}\ (\perp)\ \mid\ \mathtt{t} \geq 0\}$$

$$\mathtt{enlift(delift(lb))} = \mathtt{lb}$$

and

$$\forall\ \mathtt{b} \in \mathbb{BOOL},\ \mathtt{b} \neq <\infty,\perp>$$

$$\mathtt{delift(enlift(b))} = \mathtt{b}$$

Proposition : $\forall\ \mathtt{lb} \in \mathbb{LBOOL},$ (A1.13)

$$\mathtt{lb} \neq \bigsqcup\{\ (\lambda(\mathtt{x}).\ \mathtt{in}\mathbb{LBOOL}_{\perp}(\mathtt{lift(x)}))^{\mathtt{t}}\ (\perp)\ \mid\ \mathtt{t} \geq 0\},$$

$$\mathtt{enlift(delift(lb))} = \mathtt{lb}$$

Proof

Base cases :

$$\mathtt{delift}(\perp) = <0,\perp>$$

$$\mathtt{enlift}(<0,\perp>) = \perp$$

$$\mathtt{enlift(delift(\ in\mathbf{BOOL}(true)\ ))} = \mathtt{in\mathbf{BOOL}(true)}$$

$$\mathtt{enlift(delift(\ in\mathbf{BOOL}(false)\ ))} = \mathtt{in\mathbf{BOOL}(false)}$$

Induction step : Assume that for $\mathtt{lb} \in \mathbb{LBOOL}$, $\mathtt{enlift(delift(lb))} = \mathtt{lb}$

$$\mathtt{enlift(delift(\ in}\mathbb{LBOOL}_{\perp}\mathtt{(lift(lb))\ ))}$$

$$= \mathtt{enlift(<t"+1,lb">)}\ \mathbf{where}\ \mathtt{<t",lb">} = \mathtt{delift(lb)}$$

$$= \mathtt{in}\mathbb{LBOOL}_{\perp}\mathtt{(lift(enlift(<t",lb">)))}$$

$$\mathbf{where}\ \mathtt{<t",lb">} = \mathtt{delift(lb)}$$

$$= \mathtt{in}\mathbb{LBOOL}_{\perp}\mathtt{(\ lift(enlift(delift(lb))\ )}$$

$$= \mathtt{in}\mathbb{LBOOL}_{\perp}\mathtt{(\ lift(lb)\ )}$$ by induction hypothesis

$$\Rightarrow \forall\ \mathtt{lb} \in \mathbb{LBOOL},$$

$$\mathtt{lb} \neq \bigsqcup\{\ (\lambda(\mathtt{x}).\ \mathtt{in}\mathbb{LBOOL}_{\perp}(\mathtt{lift(x)}))^{\mathtt{t}}\ (\perp)\ \mid\ \mathtt{t} \geq 0\},$$

$$\mathtt{enlift(delift(lb))} = \mathtt{lb}$$

Proposition : $\forall <t,b> \in \mathbb{BOOL},\ <t,b> \neq <\infty,\bot>,$ (A1.14)
$\mathtt{delift(enlift(<t,v>))} = <t,v>$

Proof

Base cases :

$\mathtt{enlift}(<0,\bot>) = \bot$

$\mathtt{delift}(\bot) = <0,\bot>$

$\mathtt{delift(enlift(<0,b>))}$
$= \mathtt{delift(\ in\mathbf{BOOL}(b)\ )}$
$= <0,b>$

Induction step : Assume that for $<t,b> \in \mathbb{BOOL},\ <t,b> \neq <\infty,\bot>,$
$\mathtt{delift(enlift(<t,b>))} = <t,b>$

$\mathtt{delift(enlift(<t+1,b>))}$
$= \mathtt{delift(in}\mathbb{LBOOL}_\bot\mathtt{(lift(enlift(<t,b>))))}$
$= <t'+1,b'>$ **where** $<t',b'> = \mathtt{delift(enlift(<t,b>))}$
$= <t,b>$ by induction hypothesis

$\Rightarrow \forall <t,b> \in \mathbb{BOOL},\ <t,b> \neq <\infty,\bot>,$
$\mathtt{delift(enlift(<t,b>))} = <t,b>$

$\Rightarrow$ The finite elements of $\mathbb{BOOL}$ and $\mathbb{BOOL}$ are isomorphic.

The only infinite element of $\mathbb{BOOL}$ is $<\infty,\bot>$. The only infinite element of $\mathbb{LBOOL}$ is $\bigsqcup\{\ (\lambda(x)\ .\ \mathtt{in}\mathbb{LBOOL}_\bot(\mathtt{lift}(x)))^t\ (\bot)\ \mid\ t \geq 0\}$. The infinite element of $\mathbb{LBOOL}$ is the least upper bound of the set of spine elements (i.e. those elements of $\mathbb{LBOOL}$ formed by applying the lifting operator to $\bot$ some finite number of times).

From the ordering on $\mathbb{BOOL}$ we see that $<\infty,\bot>$ is an upper bound of the set of spine elements in $\mathbb{BOOL}$ since

$\forall\ t \in \mathrm{NUM},\ <t,\bot> \sqsubseteq <\infty,\bot>$

Moreover, it must be the least upper bound, as shown by the following :

Proposition : $\langle\infty,\bot\rangle = \bigsqcup\{\ \langle t,\bot\rangle \mid t \in \mathrm{NUM}\}$ (A1.15)

Proof

Assume : $\exists\ \langle t',\bot\rangle \in \mathbb{BOOL}$, such that

$\forall\ \langle t,\bot\rangle \in \mathbb{BOOL}\ \langle t,\bot\rangle \sqsubseteq \langle t',\bot\rangle$ and $\langle t',\bot\rangle \sqsubseteq \langle\infty,\bot\rangle$

Either

1. $t' \in \mathrm{NUM}$

   $\Rightarrow t'+1 \in \mathrm{NUM}$

   $\Rightarrow \langle t'+1,\bot\rangle \in \mathbb{BOOL}$

   $\Rightarrow \langle t',\bot\rangle \sqsubseteq \langle t'+1,\bot\rangle$ from the ordering on BOOL.

or :

2. $t' = \infty$

   $\Rightarrow \langle t',\bot\rangle = \langle\infty,\bot\rangle$

$\Rightarrow \langle\infty,\bot\rangle$ is the least upper bound of the spine elements of $\mathbb{BOOL}$.

Since the finite elements of $\mathbb{BOOL}$ and $\mathbb{LBOOL}$ are isomorphic, and since the infinite element of $\mathbb{BOOL}$ corresponds to the infinite element of $\mathbb{LBOOL}$, $\mathbb{BOOL}$ and $\mathbb{LBOOL}$ are isomorphic.

The herring-bone domain construction can be generalised to an arbitrary domain as follows :

Definition : General Herring-Bone Domains (A1.16)

Given a domain definition $\mathrm{D} = \mathrm{F(D)}$, where $\mathrm{F(D)}$ is a domain expression which may or may not refer to D, let the corresponding herring-bone domain be $\mathbb{D} = \mathrm{F}(\mathbb{D}) \oplus \mathbb{D}_\bot$

Constructors :

$\langle\_,\ \_\rangle : \mathrm{NUM} \times \mathrm{F}(\mathbb{D}) \rightarrow \mathbb{D}$

$\langle\infty,\ \bot\rangle : \qquad \rightarrow \mathbb{D}$

Selectors :

$$(\mathbf{Match}\ \langle t,v\rangle\ \mathbf{with}\ \langle t',\bot\rangle \to e_1,\ \langle\infty,\bot\rangle \to e_2,\ \langle t',fd'\rangle \to e_3)$$

such that

$$(\mathbf{Match}\ \langle t,\bot\rangle\ \mathbf{with}\ \langle t',\bot\rangle \to e_1,\ \langle\infty,\bot\rangle \to e_2,\ \langle t',fd'\rangle \to e_3)$$
$$= (\lambda(t').e_1)\ (t)$$
$$(\mathbf{Match}\ \langle\infty,\bot\rangle\ \mathbf{with}\ \langle t',\bot\rangle \to e_1,\ \langle\infty,\bot\rangle \to e_2,\ \langle t',fd'\rangle \to e_3)$$
$$= e_2$$
$$(\mathbf{Match}\ \langle t,fd\rangle\ \mathbf{with}\ \langle t',\bot\rangle \to e_1,\ \langle\infty,\bot\rangle \to e_2,\ \langle t',fd'\rangle \to e_3)$$
$$= (\lambda(t',d').e_3)\ (t,fd) \quad \text{where } fd \neq \bot$$

and :

$$\langle\!\langle\ \langle t,v\rangle\ :\ \langle t',fd'\rangle \to e\ \rangle\!\rangle$$

such that

$$\begin{array}{l}
\langle\!\langle\ \langle t,v\rangle\ :\ \langle t',fd'\rangle \to e\ \rangle\!\rangle \\
\quad = (\mathbf{Match}\ \langle t,v\rangle\ \mathbf{with} \\
\qquad\qquad \langle t',\bot\rangle \quad \to \langle t',\bot\rangle, \\
\qquad\qquad \langle t',fd'\rangle \to (\mathbf{Match}\ (\ (\lambda(t',fd').e)\ (t',fd')\ )\ \mathbf{with} \\
\qquad\qquad\qquad\qquad \langle t'',\bot\rangle \quad \to \langle max(t',t''),\bot\rangle, \\
\qquad\qquad\qquad\qquad \langle t'',v''\rangle \to \langle max(t',t''),v''\rangle \\
\qquad\qquad\qquad\quad) \\
\qquad)
\end{array}$$

and also :

$$\langle\!\langle\ \langle t,v\rangle \to e\ \rangle\!\rangle$$

such that

$$\begin{array}{l}
\langle\!\langle\ \langle t,v\rangle \to e\ \rangle\!\rangle \\
\quad = \langle\!\langle\ \langle t,v\rangle\ :\ \langle t,v\rangle \to e\ \rangle\!\rangle
\end{array}$$

A1.3 Non-Primitive Domain Definitions

This section contains the definition of the non-primitive domains used in the semantics of **Ruth**. Note that unless the ordering, constructor(s) and selector(s) are explicitly given for a domain the standard ordering, constructor(s) and selector(s) implied its defining equation are assumed.

Definition : Environments (A1.17)

Let `Id` be the syntactic domain of identifiers.

$$\mathbf{ENV} = \mathtt{Id} \rightarrow (\mathbf{UNDEF} \oplus \mathbb{VAL})$$

Constructors :

$\varnothing$	:	$\rightarrow \mathbf{ENV}$
$[\_ \rightarrow \_]$	: $\mathtt{Id} \times \mathbb{VAL}$	$\rightarrow \mathbf{ENV}$
$\_ \; \& \; \_$	: $\mathbf{ENV} \times \mathbf{ENV}$	$\rightarrow \mathbf{ENV}$

Selector : $\_[\_] \; : \; \mathbf{ENV} \times \mathtt{Id} \rightarrow \mathbf{UNDEF} \oplus \mathbb{VAL}$

such that

$\varnothing[\mathtt{I}]$

$\quad = \mathtt{in}\mathbf{UNDEF}(\mathtt{Undef})$

$[\mathtt{I} \rightarrow \mathtt{v}] \; [\mathtt{I'}]$

$\quad = (\mathtt{I} = \mathtt{I'} \rightarrow \mathtt{in}\mathbb{VAL}(\mathtt{v}), \; \mathtt{in}\mathbf{UNDEF}(\mathtt{Undef}) \;)$

$(\rho 1 \; \& \; \rho 2)[\mathtt{I}]$

$\quad = (\mathtt{Cases} \; \rho_2[\mathtt{I}] \; \mathtt{of}$

$\qquad \mathtt{is}\mathbf{UNDEF}(\mathtt{Undef}) \quad \rightarrow \rho_1[\mathtt{I}]$

$\qquad \mathtt{else} \quad \rightarrow \rho_2[\mathtt{I}]$

$\quad)$

Definition : Atoms (A1.18)

$$\mathbf{ATOM} = \mathbf{BOOL} \oplus \mathbf{INT} \oplus \mathbf{STRING}$$

Definition : Clocked Functions (A1.19)

$$\mathbf{FUNC} = \mathbb{CLK} \rightarrow \mathbf{F}$$

$$\mathbf{F} \;\; = \mathbb{VAL} \rightarrow (\mathbf{F} \oplus \mathbb{VAL})$$

Definition : Tuples (A1.20)

$\mathbf{TUPLE} = (\mathbb{ELEM}_\perp \otimes \mathbf{TUPLE}) \oplus \mathbf{NIL}$

$\mathbb{ELEM} = \mathbb{CHAN} \oplus \mathbb{ELEM}_\perp$

Constructors :

$\{\ \}\ :\ \rightarrow \mathbf{TUPLE}$

$\{\_,\ \ldots,\ \_\}\ :\ (\mathbb{ELEM} \times \ldots \times \mathbb{ELEM}) \rightarrow \mathbf{TUPLE}$

such that

$\{\ \} = \texttt{in}\mathbf{NIL}\texttt{(Nil)}$

$\{el_1, el_2, \ldots, el_m\}$

$= \texttt{in}\mathbb{ELEM}_\perp\otimes\mathbf{IOTUPLE}\texttt{([lift(}el_1\texttt{)},\{el_2,\ldots,el_m\}\texttt{])}$

Selector : Using pattern matching on $\{el_1, el_2, \ldots, el_m\}$ structures

Definition : Input/Output Tuples (A1.21)

$\mathbf{IOTUPLE} = (\mathbb{CHAN}_\perp \otimes \mathbf{IOTUPLE}) \oplus \mathbf{NIL}$

Constructors :

$\{\{\ \}\}\ :\ \rightarrow \mathbf{IOTUPLE}$

$\{\{\_,\ \ldots,\ \_\}\}\ :\ (\mathbb{CHAN} \times \ldots \times \mathbb{CHAN}) \rightarrow \mathbf{IOTUPLE}$

such that

$\{\{\ \}\} = \texttt{in}\mathbf{NIL}\texttt{(Nil)}$

$\{\{ch_1, ch_2, \ldots, ch_m\}\}$

$= \texttt{in}\mathbb{CHAN}_\perp\otimes\mathbf{IOTUPLE}\texttt{([lift(}ch_1\texttt{)},\{\{ch_2,\ldots,ch_m\}\}\texttt{])}$

Selector : Using pattern matching on $\{\{ch_1, ch_2, \ldots, ch_m\}\}$ structures

Definition : S-expressions (A1.22)

$\mathbf{S\text{-}EXP} = \mathbf{NIL} \oplus \mathbf{ATOM} \oplus \mathbf{PAIR}$

$\mathbf{PAIR} = (\mathbb{S\text{-}EXP} \times \mathbb{S\text{-}EXP})_\perp$

$\mathbb{S\text{-}EXP} = \mathbf{S\text{-}EXP} \oplus \mathbb{S\text{-}EXP}_\perp$

Definition : Clocks (A1.23)

$\mathbb{CLK} = (\mathbb{CLK} \times \mathbb{CLK}) \oplus \mathbb{CLK}_\perp$

Definition : Channels (A1.24)

$$\mathbb{CHAN} = (\mathbf{ATOM} \otimes \mathbb{CHAN}_{\perp}) \oplus \mathbb{CHAN}_{\perp}$$

Definition : Expressible values (A1.25)

$$\mathbb{VAL} = (\mathbf{S\text{-}EXP} \oplus \mathbf{FUNC} \oplus \mathbf{TUPLE} \oplus \mathbb{CLK} \oplus \mathbb{CHAN}) \oplus \mathbb{VAL}_{\perp}$$

Appendix 2 : Formal Presentation Of The Semantics of Ruth

A2.1 Syntactic Domains

Definition : The Syntactic Domains (A2.1)

Pr ∈ Prog	Programs
C ∈ Conf	Process Configurations
P ∈ Proc	Processes
PA ∈ Pr-App	Process Applications
B ∈ Bool	Booleans
N ∈ Int	Integers
Y ∈ String	Symbolic Strings
I ∈ Id	Identifiers
E ∈ Exp	Syntactic Expressions
D ∈ Dec	Declarations

A2.2 Abstract Syntax

Definition : Abstract Syntax (A2.2)

```
Pr ::= P...P C

C  ::= Configuration I Output I...I  Input I...I
       Is PA  ... PA

P  ::= Process I Input I...I Clock I
       Is E

PA ::= I...I = I (I...I)

E  ::= Y | B | N | I | E + E | E - E | E * E | E / E | E \ E |
       E = E | E ≠ E | E ≤ E | E ≥ E | E < E | E > E | Atom (E) |
       Nil | isNil (E) | Cons (E E) | Head (E) | Tail (E) |
       ConsCh (E E E) | HeadCh (E) | Time (E) | TailCh (E) |
       Ready (E E) | {E...E} | E ! E | HeadClk (E) |
       TailClk(E) | E And E | E Or E | Not E |
          If E Then E Else E | E where D endwhere |
          E whererec D endwhererec | lambda (I...I) E | E (E...E)

      D  ::= I = E D |
```

A2.3 Semantic Domains

Definition : The Semantic Domains (A2.3)

i,n	∈	**INT**	As defined by (A1.1)
t,n	∈	**NUM**	As defined by (A1.1)
b	∈	**BOOL**	As defined by (A1.1)
y	∈	**STRING**	As defined by (A1.1)
		NIL	As defined by (A1.1)
		UNDEF	As defined by (A1.1)
		NUM	As defined by (A1.1)
ρ	∈	**ENV**	As defined by (A1.17)
a	∈	**ATOM**	As defined by (A1.18)
f	∈	**FUNC**	As defined by (A1.19)
tp	∈	**TUPLE**	As defined by (A1.20)
		IOTUPLE	As defined by (A1.21)
s	∈	**S-EXP**	As defined by (A1.22)
s	∈	S-EXP	As defined by (A1.22)
c	∈	CLK	As defined by (A1.23)
ch	∈	CHAN	As defined by (A1.24)
<t,v>	∈	VAL	As defined by (A1.25)

A2.4 Semantic Functions

Definition : The Meaning Functions (A2.4)

$$\begin{array}{llllllllll}
\mathcal{E}_{Pr} & = & \texttt{Prog} & \rightarrow \mathbb{CLK} & \rightarrow \textbf{IOTUPLE} & & \rightarrow \textbf{IOTUPLE} & \\
\mathcal{E}_{C} & = & \texttt{Conf} & \rightarrow \textbf{ENV} & \rightarrow \mathbb{CLK} & & \rightarrow \textbf{IOTUPLE} & \rightarrow \textbf{IOTUPLE} \\
\mathcal{E}_{P} & = & \texttt{Proc} & \rightarrow \textbf{ENV} & & & & \\
\mathcal{E}_{PA} & = & \texttt{Pr-App} & \rightarrow \textbf{ENV} & \rightarrow \mathbb{CLK} & & \rightarrow \textbf{ENV} & \\
\mathcal{E}_{V} & = & \texttt{Exp} & \rightarrow \textbf{ENV} & \rightarrow \mathbb{CLK} & & \rightarrow \mathbb{VAL} & \\
\mathcal{E}_{D} & = & \texttt{Dec} & \rightarrow \textbf{ENV} & \rightarrow \mathbb{CLK} & & \rightarrow \textbf{ENV} & \\
\mathcal{E}_{Y} & = & \texttt{Symbol} & \rightarrow \textbf{STRING} & & & & \\
\mathcal{E}_{B} & = & \texttt{Bool} & \rightarrow \textbf{BOOL} & & & & \\
\mathcal{E}_{N} & = & \texttt{Int} & \rightarrow \textbf{INT} & & & &
\end{array}$$

In defining these functions we will find the following functions useful.

Definition : Environment lookup function (A2.5)

```
lookup : Id x ENV → 𝕍𝔸𝕃
lookup
    = λ(I,ρ) .
        (Cases ρ[I] of
          isUNDEF(Undef)        → <∞,⊥>
          is𝕍𝔸𝕃(lift(<t,v>)    → <t,v>
        )
```

Definition : Clock Extraction Function (A2.6)

```
extract : ℂ𝕃𝕂 x NUM → ℂ𝕃𝕂
extract
    = λ(c,n) .
        « c : <t,[l,r]>
              → (n ≤ 0 → from(l,t), extract(from(r,t),n-1)) )
        »

from : ℂ𝕃𝕂 x NUM → ℂ𝕃𝕂
from = λ(c,n) . « c : <$t_c$,[$l_c$,$r_c$]> → <$t_c$+n+1,[$l_c$,$r_c$]> »
```

The following convention is used with `extract` :

$\forall$ c $\in$ $\mathbb{CLK}$, n $\in$ NUM, c_n denotes `extract(c,n)`

Definition : Ageing Functions (A2.7)

```
after : $\mathbb{VAL}$ x $\mathbb{CLK}$ → $\mathbb{VAL}$
after
   = λ(v, c).
       « c : <$t_c$,[$l_c$,$r_c$]>
           → « v : <t,d>
                  → ($t_c$ > t → <$t_c$,v>, after(<t,d>,from($l_c$,$t_c$)
             »
       »

clockafter : $\mathbb{CLK}$ x NUM → $\mathbb{CLK}$
clockafter = λ(c,n). from(c,n)
```

Definition : `filter` Function (A2.8)

```
filter : $\mathbb{CHAN}$ x NUM x $\mathbb{CLK}$ → $\mathbb{CHAN}$
filter = λ(ch,n,c).<t-n,v>
         where
         <t,v> = « c : <$t_c$,[$l_c$,$r_c$]>
                      → « ch : <t,[a,lift(rest)]>
                             → ($t_c$ ≤ t + n
                                 →<t + n,[a,lift(rest')]>, <∞,⊥>
                                  where
                                  rest' = filter(rest,t+n+1,
                                                       from($l_c$,$t_c$))
                               )
                        »
                 »
```

In Appendix 1 we defined `cases ... of ...` selectors for sum domains. In certain situations we would prefer to be able to test if an element is in a particular sub-domain and project from that sub-domain directly, without using the cases notation. The functions defined overleaf allow this.

Definition : VAL sub-domain accessing functions `outX` and `checkX`. (A2.9)

```
∀ X ∈ {S-EXP, FUNC, TUPLE, CLK, CHAN}
outX   : S-EXP ⊕ FUNC ⊕ TUPLE ⊕ CLK ⊕ CHAN → X
checkX : S-EXP ⊕ FUNC ⊕ TUPLE ⊕ CLK ⊕ CHAN → BOOL

outX
  = λ(v) .
      (v = ⊥ → ⊥,
        (Cases v of
         isX(x) → x
         else   → ⊥
        )
      )

checkX
  = λ(v) .
      (v = ⊥ → ⊥,
        (Cases v of
         isX(x) → true
         else   → false
        )
      )
```

Definition : S-EXP sub-domain accessing functions `outX` and `checkX`. (A2.9)

```
∀ X ∈ {NIL, ATOM, PAIR}
outX   : S-EXP → X
checkX : S-EXP ⊕ FUNC ⊕ TUPLE ⊕ CLK ⊕ CHAN → BOOL

outX
  = λ(s) .
      (s = ⊥ → ⊥,
        (Cases s of
         isX(x) → x
         else   → ⊥
        )
      )
```

```
checkX
  = λ(v) .
      (v = ⊥ → ⊥,
        (Cases v of
         isS-EXP(s)   → (Cases s of
                           isX(x)   → true
                           else     → false
                          )
         else         → false
        )
      )
```

```
∀ X ∈ {INT, BOOL, STRING}
outX   : S-EXP → X
checkX : S-EXP ⊕ FUNC ⊕ TUPLE ⊕ CLK ⊕ CHAN → BOOL
```

```
outX
  = λ(v) .
      (v = ⊥ → ⊥,
        (Cases v of
         isATOM(a) →  (Cases a of
                        isX(x) → x
                        else   → ⊥
                       )
         else        → ⊥
        )
      )
```

```
checkX
  = λ(v) .
      (v = ⊥ → ⊥,
        (Cases v of
         isS-EXP(s)   → (Cases s of
                          isATOM(a)   → (Cases a of
                                          isX(x)   → x
                                          else     → false
                                         )
                          else        → false
                         )
         else         → false
        )
      )
```

Definition : **F** ⊕ **VAL** sub-domain accessing functions `outX` and `checkX`. (A2.10)

```
∀ X ∈ {F, VAL}
outX   : F ⊕ VAL → X
checkX : F ⊕ VAL → BOOL
```

```
outX
  = λ(v) .
      (v = ⊥ → ⊥,
        (Cases v of
         isX(x) → x
         else   → ⊥
        )
      )
```

```
checkX
  = λ(v) .
      (v = ⊥ → ⊥,
        (Cases v of
         isX(x) → true
         else   → false
        )
      )
```

Definition : Maximum function (A2.11)

$$\text{max} : \textbf{INT} \times \textbf{INT} \rightarrow \textbf{INT}$$

$$\text{max} = \lambda(i_1, i_2) . (i_1 > i_2 \rightarrow i_1, i_2)$$

Note that `max` generalises to any number of arguments as below.

$$\text{max}\ (i_1, i_2, \ldots, i_m) = \text{max}\ (i_1, \text{max}\ (i_2, \ldots, \text{max}\ (i_{m-1}, i_m) \ldots))$$

For completeness we also include the definition of the Least Fixed Point operator $\mathtt{fix}$.

Definition : The Fixed Point operator $\mathtt{fix}$ (A2.12)
For any domain D the least fixed point of the continuous functional $F : D \to D$ exists and is defined to be $\mathtt{fix}(F)$ where

$$\mathtt{fix} : (D \to D) \to (D \to D)$$

$$\begin{aligned}
&\mathtt{fix}\\
&\quad = \lambda(F).\bigsqcup\{F^n(\bot) \mid n \geq 0\}\\
&\qquad \textbf{where}\\
&\qquad F^0(x) = x\\
&\qquad F^{n+1}(x) = F(F^n(x))
\end{aligned}$$

A2.5 Semantic Equations

The usual convention is followed :

$\forall\ c \in \mathbb{CLK},\ n \in \mathrm{NUM},\ \rho \in \mathbf{ENV},\ E_n \in \mathrm{Exp},\ \langle t_n, v_n\rangle$ denotes $\mathcal{E}_V \llbracket E_n \rrbracket\ \rho\ c_n$

$$\begin{aligned}
&\mathcal{E}_{Pr}\ \llbracket\ P_1 \ldots P_k\ C\ \rrbracket\quad c\ \{\{ch_1, \ \ldots\ ch_n\}\} \qquad \text{(A2.13)}\\
&\quad = \mathcal{E}_C\llbracket C\rrbracket\ \rho\ c\ \{\{ch_1, \ \ldots\ ch_n\}\}\\
&\qquad \textbf{where}\\
&\qquad \rho = \mathcal{E}_P\llbracket P_1\rrbracket\ \&\ldots\&\ \mathcal{E}_P\llbracket P_k\rrbracket
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_C\ \llbracket\ \textbf{Configuration}\ I\ \textbf{Output}\ I^1 \ldots I^m\ \textbf{Input}\ I_1 \ldots I_n \qquad \text{(A2.14)}\\
&\quad \textbf{Is}\ Pa_1 \ldots Pa_k\ \rrbracket\ \rho\ c\ \{\{ch_1,\ \ldots,\ ch_n\}\}\\
&\quad = \{\{ch^1,\ \ldots,\ ch^m\}\}\\
&\qquad \textbf{where}\\
&\qquad ch^j = \ll \mathrm{lookup}(I^j, \rho_c)\ :\ \langle t, v\rangle\\
&\qquad\qquad\qquad \to (\mathrm{check}\mathbb{CHAN}(v) \to \mathrm{out}\mathbb{CHAN}(v),\ \langle\infty, \bot\rangle)\\
&\qquad\qquad \gg\\
&\qquad \rho_c = \mathtt{fix}\ (\lambda\rho''.\rho'\ \&\ \mathcal{E}_{PA}\llbracket Pa_0\rrbracket\ \rho''\ c_0\ \&\ \ldots\ \&\ \mathcal{E}_{PA}\llbracket Pa_k\rrbracket\ \rho''\ c_k\\
&\qquad \rho' = \rho\ \&\ [I_1 \to \langle 0, \mathrm{in}\mathbb{CHAN}(ch_1)\rangle]\ \&\ldots\&\\
&\qquad\qquad\qquad [I_n \to \langle 0, \mathrm{in}\mathbb{CHAN}(ch_n)\rangle]
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_{P}\,[\![\ \textbf{Process}\ \mathtt{I}\ \textbf{Input}\ \mathtt{I}_1\ldots\mathtt{I}_n\ \textbf{Clock}\ \ \mathtt{I}_c\ \textbf{Is}\ \mathtt{E}\ ]\!] && \text{(A2.15)}\\
&\quad = [\mathtt{I} \rightarrow \mathtt{<0,f>}]\\
&\qquad \textbf{where}\\
&\qquad \mathtt{f} = \mathtt{in}\textbf{FUNC}(\lambda\delta_{sc}.\mathtt{in}\textbf{F}(\lambda\delta_{0}.\ \ldots\ \mathtt{in}\textbf{F}(\lambda\delta_{pc}.\\
&\qquad\qquad\quad \mathtt{in}\mathbb{VAL}(\mathcal{E}_V[\![\mathtt{E}]\!]\ \ ([\mathtt{I}_1 \rightarrow \delta_1]\ \&\ \ldots\ \&\ [\mathtt{I}_n \rightarrow \delta_n]\ \&\\
&\qquad\qquad\qquad\qquad [\mathtt{I}_c \rightarrow \delta_{pc}])\ \delta_{sc}\ )\ )\ldots))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_{PA}\,[\![\ \mathtt{I}^1\ldots\mathtt{I}^m = \mathtt{I}\ \ (\mathtt{I}_1\ldots\mathtt{I}_n)\ ]\!]\ \rho\ \mathtt{c} && \text{(A2.16)}\\
&\quad = [\mathtt{I}^1 \rightarrow \mathtt{<0,\ in}\mathbb{CHAN}\mathtt{(ch^1)>}]\ \&\ \ldots\&\ [\mathtt{I}^m \rightarrow \mathtt{<0,in}\mathbb{CHAN}\mathtt{(ch^m)>}]\\
&\qquad \textbf{where}\\
&\qquad \mathtt{ch}^j = \ll \mathtt{<t_j,ch_j>}\\
&\qquad\qquad\quad \rightarrow \mathtt{filter(out}\mathbb{CHAN}\mathtt{(ch_j),0,clockafter(c_j,max(t',t_j)))}\\
&\qquad\quad \gg\\
&\qquad \mathtt{<t',\ in}\textbf{TUPLE}\mathtt{(\{<t_1,ch_1>,\ \ldots,\ <t_m,ch_m>\})>}\\
&\qquad\quad = \ll \mathtt{lookup(I,}\rho\mathtt{)\ \ :\ <t,v>}\\
&\qquad\qquad\quad \rightarrow \mathtt{(check}\textbf{FUNC}\mathtt{(v)}\\
&\qquad\qquad\qquad \rightarrow \mathtt{out}\mathbb{VAL}\mathtt{(out}\textbf{F}\mathtt{(\ldots\ out}\textbf{F}\mathtt{(\ f\ i_1\ )\ \ldots\ i_n\ )\ pc\ ),}\\
&\qquad\qquad\qquad\quad \mathtt{<t,\{<}\infty\mathtt{,}\bot\mathtt{>,\ \ldots,\ <}\infty\mathtt{,}\bot\mathtt{>\}>}\\
&\qquad\qquad\qquad\quad \textbf{where}\\
&\qquad\qquad\qquad\quad \mathtt{i_k = lookup(I_k,}\rho\mathtt{)}\\
&\qquad\qquad\qquad\quad \mathtt{f = out}\textbf{FUNC}\mathtt{(v_1)\ \ extract(c_0,0)}\\
&\qquad\qquad\qquad\quad \mathtt{pc = in}\mathbb{CLK}\mathtt{(<0,extract(c_0,1)>)}\\
&\qquad\qquad\quad)\\
&\qquad\quad \gg
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_{V}\,[\![\ \mathtt{Y}\ ]\!]\ \rho\ \mathtt{c} && \text{(A2.17)}\\
&\quad = \mathtt{after(<0,in}\textbf{S-EXP}\mathtt{(in}\textbf{ATOM}\mathtt{(in}\textbf{STRING}\mathtt{(}\mathcal{E}_Y[\![\mathtt{Y}]\!]\mathtt{)))>,\ \ c)}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_{V}\,[\![\ \mathtt{B}\ ]\!]\ \rho\ \mathtt{c} && \text{(A2.18)}\\
&\quad = \mathtt{after(<0,in}\textbf{S-EXP}\mathtt{(in}\textbf{ATOM}\mathtt{(in}\textbf{BOOL}\mathtt{(}\mathcal{E}_B[\![\mathtt{B}]\!]\mathtt{)))>,\ \ c)}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_{V}\,[\![\ \mathtt{N}\ ]\!]\ \rho\ \mathtt{c} && \text{(A2.19)}\\
&\quad = \mathtt{after(<0,in}\textbf{S-EXP}\mathtt{(in}\textbf{ATOM}\mathtt{(in}\textbf{INT}\mathtt{(}\mathcal{E}_N[\![\mathtt{N}]\!]\mathtt{)))>,\ \ c)}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_{V}\,[\![\ \mathtt{I}\ ]\!]\ \rho\ \mathtt{c} && \text{(A2.20)}\\
&\quad = \mathtt{after(lookup(I,}\rho\mathtt{),\ \ c)}
\end{aligned}$$

```
$\mathcal{E}_V$ ⟦ $E_1$ + $E_2$ ⟧ ρ c                                    (A2.21)
   = after (
     ≪ <$t_1$,$v_1$
        → ≪ <$t_2$,$v_2$>
              → (checkINT($v_1$) ∧ checkINT($v_2$)
                   → <max($t_1$,$t_2$),inS-EXP(inATOM(inINT(res)))>,
                     <∞,⊥>
                )
           ≫
     ≫,
     $c_0$)
     where
     res = outINT(outATOM(outS-EXP($v_1$))) +
           outINT(outATOM(outS-EXP($v_2$)))
```

The definitions of - (A2.22), * (A2.23), / (A2.24) and \ (A2.25) can be obtained by substituting the relevant operator for + in the definition of res in (A2.21).

```
$\mathcal{E}_V$ ⟦ $E_1$ = $E_2$ ⟧ ρ c                                    (A2.26)
   = after (
     ≪ <$t_1$,$v_1$>
        → ≪ <$t_2$,$v_2$>
              → (checkATOM($v_1$) ∧ checkATOM($v_2$)
                   → <max($t_1$,$t_2$),inS-EXP(inATOM(inBOOL(res)))>,
                     <∞,⊥>
                )
           ≫
     ≫,
     $c_0$)
     where
     res = outATOM(outS-EXP($v_1$)) = outATOM(outS-EXP($v_2$)))
```

The definition of ≠ (A2.27) can be obtained by substituting ≠ for = in the definition of res in (A2.26).

```
$\mathcal{E}_V [\![ E_1 \le E_2 ]\!] \rho\ c$                                    (A2.28)
  = after (
    $\ll \langle t_1, v_1 \rangle$
       $\rightarrow \ll \langle t_2, v_2 \rangle$
             $\rightarrow$ (check$\mathbf{INT}(v_1) \wedge$ check$\mathbf{INT}(v_2)$
                   $\rightarrow \langle \max(t_1, t_2)$, in$\mathbf{S\text{-}EXP}$(in$\mathbf{ATOM}$(in$\mathbf{BOOL}$(res)))$\rangle$,
                     $\langle \infty, \perp \rangle$
               )
          $\gg$
    $\gg$,
    $c_0$)
    where
    res = out$\mathbf{INT}$(out$\mathbf{ATOM}$(out$\mathbf{S\text{-}EXP}(v_1)$)) $\le$
          out$\mathbf{INT}$(out$\mathbf{ATOM}$(out$\mathbf{S\text{-}EXP}(v_2)$))
```

The definitions of ≥ (A2.29), < (A2.30) and > (A2.31) can be obtained by substituting the relevant operator for ≤ in the definition of res in (A2.28).

```
$\mathcal{E}_V [\![$ ATOM $(E_1)$ $]\!] \rho\ c$                                   (A2.32)
  = after (
    $\ll \langle t_1, v_1 \rangle$
         $\rightarrow$ (check$\mathbf{S\text{-}EXP}(v_1)$
              $\rightarrow \langle t_1$, in$\mathbf{S\text{-}EXP}$(in$\mathbf{ATOM}$(in$\mathbf{BOOL}$(check$\mathbf{ATOM}(v_1)$)))$\rangle$,
                $\langle \infty, \perp \rangle$
           )
    $\gg$,
    $c_0$)
```

The definition of **IsNil** (A2.33) can be obtained by substituting check**Nil** for check**Atom** in (A2.32).

```
$\mathcal{E}_V [\![$ Nil $]\!] \rho\ c$                                          (A2.34)
  = after($\langle 0$, in$\mathbf{S\text{-}EXP}$(in$\mathbf{NIL}$(Nil))$\rangle$, c)
```

```
$\mathcal{E}_V$ ⟦ Cons (E$_1$ E$_2$) ⟧ ρ c                                    (A2.35)
   = after(<0,inS-EXP(inPAIR(lift([s$_1$,s$_2$])))>,c$_0$)
     where
     s$_1$ = ≪ <t$_1$,v$_1$>
              → (checkS-EXP(v$_1$) → <t$_1$,outS-EXP(v$_1$)>, <∞,⊥>)
          ≫
     s$_2$ = ≪ <t$_2$,v$_2$>
              → (checkS-EXP(v$_2$) → <t$_2$,outS-EXP(v$_2$)>, <∞,⊥>)
          ≫


$\mathcal{E}_V$ ⟦ Head (E$_1$) ⟧ ρ c                                          (A2.36)
   = after (
     ≪ <t$_1$,v$_1$>
          → (checkS-EXP(v$_1$)
              → (Cases outS-EXP(v$_1$) of
                   isPAIR(lift([<t$_H$,s$_H$>,<t$_T$,s$_T$>]))
                    → <max(t$_1$,t$_H$),inS-EXP(s$_H$)>
                   else → <∞,⊥>
                 ),
                 <∞,⊥>
            )
     ≫,
     c$_0$)


$\mathcal{E}_V$ ⟦ Tail (E$_1$) ⟧ ρ c                                          (A2.37)
   = after (
     ≪ <t$_1$,v$_1$>
          → (checkS-EXP(v$_1$)
              → (Cases outS-EXP(v$_1$) of
                   isPAIR(lift([<t$_H$,s$_H$>,<t$_T$,s$_T$>]))
                    →          <max(t$_1$,t$_T$),inS-EXP(s$_T$)>
                   else → <∞,⊥>
                 ),
                 <∞,⊥>
            )
     ≫,
     c$_0$)
```

```
$\mathcal{E}_V [\![ \mathbf{ConsCh}(E_1, E_2, E_3) ]\!] \rho\ c$                                    (A2.38)
  = after (
    « $<t_1,v_1>$
       → « $<t_2,v_2>$
              → (check$\mathbf{ATOM}(v_1) \wedge$ check$\mathbf{INT}(v_2)$
                     → $(i_2 \le t \to <\infty,\bot>, <t,ch>), <\infty,\bot>$
                 )
          »
    »,
    $c_0$)
    where
    $a_1$ = out$\mathbf{ATOM}$(out$\mathbf{S\text{-}EXP}(v_1)$))
    $i_2$ = out$\mathbf{INT}$(out$\mathbf{ATOM}$(out$\mathbf{S\text{-}EXP}(v_2)$)))
    <t,ch> = after(<max$(t_1,t_2)$,in$\mathbb{CHAN}$($<i_2,[a_1$,lift(rest)]>)>,$c_0$)
    rest
     = <t"-$(i_2+1)$,v")>
       where
       <t",v"> =  « $<t_3,v_3>$
                       → (check$\mathbb{CHAN}(v_3)$
                            → « out$\mathbb{CHAN}(v_3)$  : <t',[a',lift(ch')]>
                                  → (t'$\ge t_3 \wedge$ t' > $i_2$
                                       → <t',[a',lift(ch')]>,$<\infty,\bot>$
                                     )
                               »,
                               $<\infty,\bot>$
                          )
                  »


$\mathcal{E}_V [\![ \mathbf{HeadCh}\ (E_1) ]\!] \rho\ c$                                      (A2.39)
  = after (
    « $<t_1,v_1>$
         → (check$\mathbb{CHAN}(v_1)$
              → (« out$\mathbb{CHAN}(v_1)$  : <t,[a,lift(ch)]>
                     → <max$(t_1$,t),in$\mathbf{S\text{-}EXP}$(in$\mathbf{ATOM}$(a))>
                 »
                ),
                $<\infty,\bot>$
           )
    »,
    $c_0$)
```

```
$\mathcal{E}_V$ ⟦ Time (E_1) ⟧ ρ c                                          (A2.40)
   = after (
     « <t_1,v_1>
           → (checkCHAN(v_1)
                 → (« outCHAN(v_1)  : <t,[a,lift(ch)]>
                              → <max(t_1,t),inS-EXP(inATOM(inINT(t)))>
                     »
                    ),
                    <∞,⊥>
              )
     »,
     c_0)


$\mathcal{E}_V$ ⟦ TailCh (E_1) ⟧ ρ c                                        (A2.41)
   = after (
     « <t_1,v_1>
           → (checkCHAN(v_1)
                →« outCHAN(v_1)  : <t,[a,lift(ch)]>
                     → <max(t_1,t), inCHAN(<t'+t+1,v'>)>
                       where
                       <t'v'> =  « ch : <t_r,[a_r,lift(ch_r)]>
                                          → <t_r,[a_r,lift(ch_r)]>
                                 »
                 »,
                <∞,⊥>
              )
     »,
     c_0)
```

```
$\mathcal{E}_V [\![$ Ready $(E_1, E_2)$ $]\!]$ $\rho$ c                                  (A2.42)
  = after (
    « $<t_1,v_1>$
        → « $<t_2,v_2>$
              →  (checkCHAN($v_1$) ∧ checkINT($v_2$)
                   →(Match outCHAN($v_1$) with
                       <t,⊥>
                         → (t > $i_2$
                              → <max($t_1$,$t_2$,$i_2$),ff>,
                                <max($t_1$,$t_2$,t),⊥>
                           )
                       <t,[a,lift(ch)]>
                         → (t > $i_2$
                              → <max($t_1$,$t_2$,$i_2$),ff>,
                                <max($t_1$,$t_2$,t),tt>
                           ),
                     ),
                     <∞,⊥>
                 )
          »
    »,
    $c_0$)
    where
    $i_2$ = outINT(outATOM(outS-EXP($v_2$)))
    tt = inS-EXP(inATOM(inBOOL(true)))
    ff = inS-EXP(inATOM(inBOOL(false)))
```

```
$\mathcal{E}_V [\![$ $\{E_1... E_n\}$ $]\!]$ $\rho$ c                                   (A2.43)
  = after(<0,inTUPLE({$<t^1,ch^1>$, ..., $<t^n,ch^n>$})>, $c_0$)
    where
    $<t^k,ch^k>$ =  « $<t_k,v_k>$ :<t,v>
                       →  (checkCHAN(v) → <t,outCHAN(v)>,<∞,⊥>)
                 »
```

```
$\mathcal{E}_V [\![ E_1 \; ! \; E_2 ]\!] \rho \; c$                                        (A2.44)
   = after (
     $\langle\!\langle \; \langle t_1, v_1 \rangle$
          $\rightarrow \; \langle\!\langle \; \langle t_2, v_2 \rangle \; : \; \langle t_2, i \rangle$
                $\rightarrow$ (check$\mathbf{TUPLE}(v_1) \wedge$ check$\mathbf{INT}(v_2)$
                      $\rightarrow \; (1 \le i \le m$
                            $\rightarrow \langle \max(t_1, t_2, t^i), ch^i \rangle, \langle \infty, \bot \rangle),$
                              $\langle \infty, \bot \rangle$
                              **where**
                              $\{\langle t^1, ch^1 \rangle, \; \ldots, \; \langle t^m, ch^m \rangle\} = v_1$
                        )
                  )
            $\rangle\!\rangle$
     $\rangle\!\rangle$,
     $c_0$)

$\mathcal{E}_V [\![$ **HeadClk** $(E_1) \; ]\!] \rho \; c$                                 (A2.45)
   = after (
     $\langle\!\langle \; \langle t_1, v_1 \rangle$
          $\rightarrow$ (check$\mathbb{CLK}(v_1)$
                $\rightarrow \; \langle\!\langle$ out$\mathbb{CLK}(v_1) \; : \; \langle t, [l, r] \rangle$
                      $\rightarrow \langle \max(t_1, t)$, in$\mathbf{S\text{-}EXP}$(in$\mathbf{ATOM}$(in$\mathbf{INT}(t))) \rangle$
                  $\rangle\!\rangle$,
                  $\langle \infty, \bot \rangle$
            )
     $\rangle\!\rangle$,
     $c_0$)

$\mathcal{E}_V [\![$ **TailClk** $(E_1) \; ]\!] \rho \; c$                                 (A2.46)
   = after (
     $\langle\!\langle \; \langle t_1, v_1 \rangle$
        $\rightarrow$ (check$\mathbb{CLK}(v_1)$
              $\rightarrow \; \langle\!\langle$ out$\mathbb{CLK}(v_1) \; : \; \langle t_c, [l_c, r_c] \rangle$
                    $\rightarrow \langle \max(t_1, t_c)$, from$(l_c, t_c) \rangle$
                $\rangle\!\rangle$,
           $\langle \infty, \bot \rangle$
          )
     $\rangle\!\rangle$,
     $c_0$)
```

```
$\mathcal{E}_V$ ⟦ $E_1$ And $E_2$ ⟧ ρ c                                  (A2.47)
   = after (
     « <$t_1$,$v_1$>
         → « <$t_2$,$v_2$>
                → (checkBOOL($v_1$) ∧ checkBOOL($v_2$)
                       → <max($t_1$,$t_2$),inS-EXP(inATOM(inBOOL(res)))>,
                         <∞,⊥>
                   )
           »
     »,
     $c_0$)
     where
     res = outBOOL(outATOM(outS-EXP($v_1$))) ∧
           outBOOL(outATOM(outS-EXP($v_2$)))
```

The definition of Or (A2.48) can be obtained by substituting ∨ for ∧ in the definition of res in (A2.47).

```
$\mathcal{E}_V$ ⟦ Not $E_1$ ⟧ ρ c                                        (A2.49)
   = after (
     « <$t_1$,$v_1$>
            → (checkS-EXP($v_1$)
                  → <$t_1$,inS-EXP(inATOM(inBOOL(res)))>, <∞,⊥>
              )
     »,
     $c_0$)
     where
     res = ¬ outBOOL(outATOM(outS-EXP($v_1$)))
```

```
$\mathcal{E}_V [\![ \textbf{If } E_1 \textbf{ Then } E_2 \textbf{ Else } E_3 ]\!] \rho\ c$                                    (A2.50)
   $= \langle\!\langle\ <t_1,v_1>$
         $\rightarrow$ (check$\textbf{BOOL}(v_1)$
              $\rightarrow$ (b $\rightarrow$ $\mathcal{E}_V [\![E_2]\!] \rho$ clockafter$(c_2,t_1)$,
                        $\mathcal{E}_V [\![E_3]\!] \rho$ clockafter$(c_3,t_1)$
                 ),
               $<\infty,\perp>$
           ),
     $\rangle\!\rangle$,
     where
     $<t_1,v_1>$ = after($\mathcal{E}_V [\![E_1]\!] \rho\ c_1,\ c_0$)
     b          = out$\textbf{BOOL}$(out$\textbf{ATOM}$(out$\textbf{S-EXP}(v_1)$))
```

```
$\mathcal{E}_V [\![ \text{E } \textbf{where} \text{ D } \textbf{endwhere} ]\!] \rho\ c$                                   (A2.51)
   $= \mathcal{E}_V[\![E]\!]\ (\mathcal{E}_D[\![D]\!]\ \rho\ c_1\ )\ c_0$
```

```
$\mathcal{E}_V [\![ \text{E } \textbf{wherere c} \text{ D } \textbf{endwhererec} ]\!] \rho\ c$                             (A2.52)
   $= \mathcal{E}_V[\![E]\!]\ \rho'\ c_0$
     where
     $\rho'$ = fix($\lambda\rho''.\rho\ \&\ \mathcal{E}_D[\![D]\!]\ \rho''\ c_1$)
```

```
$\mathcal{E}_V [\![ \textbf{lambda}\ (I_0 \ldots I_n)\ E\ ]\!] \rho\ c$                                     (A2.53)
   = after(<0,f>, c)
     where
     f = in$\textbf{FUNC}(\lambda\delta_c.$in$\textbf{F}(\lambda\delta_0.$ ... in$\textbf{F}(\lambda\delta_n.$
             in$\mathbb{VAL}(\mathcal{E}_V[\![E]\!]\ (\rho\ \&\ [I_0 \rightarrow \delta_0]\ \&\ \ldots\ \&\ [I_n \rightarrow \delta_n])\ \delta_c\ )$
        )...))
```

```
$\mathcal{E}_V [\![ E_1\ (E_2 \ldots E_n)\ ]\!] \rho\ c$                                         (A2.54)
   $= \langle\!\langle\ <t_1,v_1>$
         $\rightarrow$ (check$\textbf{FUNC}(v_1)$
              $\rightarrow$out$\mathbb{VAL}$(out$\textbf{F}$(... out$\textbf{F}$( f $<t_2,v_2>$ )...) $<t_n,v_n>$),
                $<\infty,\perp>$
           )
     $\rangle\!\rangle$,
     where
     $<t_1,v_1>$  = after($\mathcal{E}_V [\![E_1]\!] \rho\ c_1,\ c_0$)
     f          = out$\textbf{FUNC}(v_1)$ clockafter$(c_0,t_1)$
```

$$\begin{aligned}&\mathcal{E}_{D} [\![\mathrm{I} = \mathrm{E}\ \mathrm{D}]\!]\ \rho\ c \\ &\quad = [\mathrm{I} \rightarrow \mathcal{E}_{V}[\![\mathrm{E}]\!]\ \rho\ c_{0}]\ \&\ \mathcal{E}_{D}[\![\mathrm{D}]\!]\ \rho\ c_{1}\end{aligned} \tag{A2.55}$$

$$\begin{aligned}&\mathcal{E}_{D} [\![\ ]\!]\ \rho\ c \\ &\quad = \varnothing\end{aligned} \tag{A2.56}$$

Appendix 3 : The Syntax of Ruth

A3.1 Introduction

This appendix gives the syntax of **Ruth** in Extended Backus-Naur Formalism (EBNF). The syntactic rules are of the following form

```
Non_Terminal ::= Syntactic_Expression
```

"|" is used to seperate alternatives in Syntactic Expressions, "[" and "]" enclose an optional term and "{" and "}" enclose a term which may be repeated any number of times (including none). Terminal symbols will be enclosed by " and ".

A3.2 Syntactic Equations

```
Program
  ::= Process_Definition ";" { Process_Definition ";" } Configuration

Process_Definition
  ::= "Process" Process_Name
      "Input" Chan_List
      "Clock" Ident
      "Is" Process_Expression

Configuration
  ::= "Configuration" Ident
      "Output" Chan_List
      "Input" Chan_List
      "Is" Process_Application ";" { Process_Application ";" }
      "end."

Process_Application
  ::= Chan_List "=" Process_Name [ "(" Chan_List ")" ]

Process_Expression
  ::= Expression

Expression
  ::= Lambda_Exp | Function_Application | If_Exp | Tuple_Exp |
      Expression "where" Definition_List "endwhere" |
      Expression "whererec" Definition_List "endwhererec" |
      Simple_Exp
```

```
Lambda_Exp
  ::= "lambda" "(" Ident_List ")" "." Expression

Function_Application
  ::= Predefined_Function_Application |
      User_Defined_Function_Application

If_Exp
  ::= "If" Expression "Then" Expression "Else" Expression

Tuple_Exp
  ::= "{" [ Exp_List ] "}"

Simple_Exp
  ::= Basic_Exp [ Rel_Op Basic_Exp ]

Definition_List
  ::= Definition ";" { Definition_List ";" }

Definition
  ::= Ident "=" Expression

Predefined_Function_Application
  ::=  "Atom" "(" Expression ")" |
       "isNil" "(" Expression ")" |
       "Cons" "(" Expression "," Expression ")" |
       "Head" "(" Expression ")" |
       "Tail" "(" Expression ")" |
       "ConsCh" "(" Expression "," Expression "," Expression ")" |
       "HeadCh" "(" Expression ")" |
       "Time" "(" Expression ")" |
       "TailCh" "(" Expression ")" |
       "Ready" "(" Expression "," Expression ")" |
       "HeadClk" "(" Expression ")" |
       "TailClk" "(" Expression ")"

User_Defined_Function_Application
  ::= Function_Name "(" Exp_List ")"

Basic_Exp
  ::= ["+" | "-"] term { Add_Op term }

term
  ::= factor { Mul_Op factor }

factor
  ::= Atom | "(" Expression ")" | "Not" factor | "Nil"

Atom
  ::= Number | Boolean | " ' " Symbol " ' "

Number
  ::= Digit {Digit}

Boolean
  ::= true | false

Symbol
  ::= Character {Character}

Chan_List
  ::= Chan_Name { "," Chan_Name }
```

```
Exp_List
  ::= Expression { "," Expression }

Ident_List
  ::= Ident { "," Ident }

Ident
  ::= Alpha { Alpha_Numeric | "_" }

Chan_Name
  ::= Ident

Process_Name
  ::= Ident

Function_Name
  ::= Ident

Alpha
  ::= "a" ... "z" |  "A"..."Z"

Digit
  ::= "0"..."9"

Character
  ::= Any printable ASCII Character

Alpha_Numeric
  ::= Alpha | Digit

Rel_Op
  ::= "<" | "≤" | ">" | "≥" | "=" | "≠"

Add_Op
  ::= "+" | "-" | "Or"

Mul_Op
  ::= "*" | "/" | "\" | "!" | "And"
```

Appendix 4 : **The Minesweep Program**

```
Process Validate_Process
Input keyboard
Clock c
Is { Time_Check (output, c) }
whererec
output     = Validate (keyboard, start_pos, first_move) ;

start_pos  = 1 ; first_move = 0 ;
comp_delay = 5 ; interval   = 200 ;

Time_Check
  = lambda (output, c) .
     If out_time ≥ soonest
     Then ConsCh (out_data, out_time,
                  Time_Check (Tail(output), TailClk(c)))
     Else ConsCh (out_data, soonest,
                  Time_Check (Tail(output), TailClk(c)))
     whererec
     out_time     = Tail(Head(output)) ;
     out_data     = Head(Head(output)) ;
     soonest      = HeadClk (c) + check_delay ;
     check_delay  = 1 ;
     endwhererec ; -- Time_Check

Validate
   = lambda (kb, pos, next_move) .
       If new_pos = pos
       Then Validate (TailCh(kb), pos, move_time + interval)
       Else Cons (Cons(new_pos, move_time),
                  Validate(TailCh(kb),new_pos,move_time+interval))
       whererec
       new_pos   = Check_and_Move (move, pos) ;
       move      = HeadCh(kb);
       move_time = If Time (kb) + comp_delay < next_move
                   Then next_move
                   Else Time (kb) + comp_delay ;
       endwhererec ; -- Validate

Check_and_Move
   = lambda (m, pos) .
       If m = 'u' And y > 1
       Then ((y-2) * array_width) + x
       Else If move = 'd' And y < array_height
            Then (y * array_width) + x
            Else If m = 'l' And x > 1
                 Then ((y-1) * array_width) + x-1
                 Else If m = 'r' And x < array_width
                      Then ((y-1) * array_width) + x+1
                      Else pos
       whererec
       x            = ((pos-1) \ array_width) + 1;
       y            = ((pos-1) / array_width) + 1;
       array_width  = 2;
       array_height = 2;
       endwhererec ; -- Check_and_Move

endwhererec ;        -- Validate_Process
```

```
Process Cell_Process
Input rnd
Clock c
Is { Time_Check (output, c) }
whererec
output = Cell (rnd, 0, init_state, init_danger, init_period);

Time_Check
  = lambda (output, c).
     If out_time ≥ soonest
     Then ConsCh (out_data, out_time,
                  Time_Check (Tail(output), TailClk(c)))
     Else Time_Check (Tail(output), TailClk(c))
     whererec
     out_time      = Tail(Head(output)) ;
     out_data      = Head(Head(output)) ;
     soonest       = HeadClk (c) + check_delay ;
     check_delay   = 250 ;
     endwhererec ; -- Time_Check

init_state    = null ;

init_danger = 50   ;  limit_danger = 400 ;  inc_danger = 5 ;
init_period = 20000;  limit_period = 2500;  dec_period = 250 ;

null = -1 ;  mine = -2 ;

Cell
  = lambda (rnd, last_out, state, danger, period).
     If new_state = state
     Then Cell (TailCh(TailCh(rnd)), out_time, state, new_danger,
                new_period)
     Else Cons (Cons(new_state, out_time),
                Cell (TailCh(TailCh(rnd)), out_time, new_state,
                      new_danger, new_period))
     whererec
     out_time      = last_out + period +
                     ((HeadCh(rnd) - 500) * period) / 1000 ;
     new_state     = Calculate_State(HeadCh(TailCh(rnd)),danger) ;
     new_period    = If (period - dec_period) ≤ limit_period
                     Then limit_period
                     Else period - dec_period ;
     new_danger    = If (danger + inc_danger) ≥ limit_danger
                     Then limit_danger
                     Else danger + inc_danger ;
     endwhererec ; -- Cell

Calculate_State
  = lambda (n,danger).
     If n > 500
     Then n \ 10
     Else If n < danger
          Then mine
          Else null ;

endwhererec ; -- Cell_Process
```

```
Process Monitor_Process
Input f_cell_1, f_cell_2, f_cell_3, f_cell_4, f_val
Clock c
Is { Time_Check (output, c) }
whererec
output = Monitor (inputs, init_state, init_player_pos, init_score) ;

init_score      = 0  ; init_player_pos = 1 ;

init_state      = Cons(null,Cons(null,Cons(null,Cons(null,Nil)))) ;
null            = -1 ; mine = -2 ;


Time_Check
  = lambda (output, c).
     If out_time ≥ soonest
     Then ConsCh (out_data, out_time,
                  Time_Check (Tail(output), TailClk(c)))
     Else ConsCh (-1, soonest, stop)
     whererec
     out_time     = Tail(Head(output) ;
     out_data     = Head(Head(output) ;
     soonest      = HeadClk (c) + check_delay ;
     stop         = ConsCh (-1, 0, stop) ;
     check_delay  = 10 ;
     endwhererec ; -- Time_Check


inputs = Scan_and_Sort (f_cell_1, f_cell_2, f_cell_3, f_cell_4,
                        f_val, init_scan)
         whererec
         init_scan     = 200 ;
         scan_interval = 200 ;
         Scan_and_Sort
          = lambda (f_cell_1, f_cell_2, f_cell_3, f_cell_4, f_val,
                    scan_time).
             Append (Quicksort (event_list, Nil),
                     Scan_and_Sort (new_f_cell_1, new_f_cell_2,
                                    new_f_cell_3, new_f_cell_4,
                                    new_f_val,
                                    scan_time + scan_interval) )
             whererec
             event_list
              = Get_Events ({f_cell_1, f_cell_2, f_cell_3,
                             f_cell_4, f_val}, 5, scan_time)) ;

             new_f_cell_1 = New_Chan (f_cell_1, check_time) ;
             new_f_cell_2 = New_Chan (f_cell_2, check_time) ;
             new_f_cell_3 = New_Chan (f_cell_3, check_time) ;
             new_f_cell_4 = New_Chan (f_cell_4, check_time) ;
             new_f_val    = New_Chan (f_val,    scan_time) ;
             endwhererec ; -- Scan_and_Sort

         Append = lambda (l1, l2).
                   If l1 = Nil
                   Then l2
                   Else Cons(Head(l1), Append (Tail(l1), l2)) ;
```

```
Quicksort
 = lambda (evl1, evl2).
    If isNil(evl1)
    Then evl2
    Else Quicksort(sooner,
                   Cons(Head(evl1),Quicksort (later,evl2))
    whererec
    sooner = Sooner(Tail(Tail(Head(evl1))), Tail(evl1)) ;
    later  = Later (Tail(Tail(Head(evl1))), Tail(evl1)) ;
    Sooner
     = lambda (time, evl).
        If isNil(evl)
        Then Nil
        Else If time < Tail(Tail(Head(evl)))
             Then Sooner(time, Tail(evl))
             Else Cons(Head(evl),Sooner(time,Tail(evl)));
    Later
     = lambda (time, evl).
        If isNil(evl)
        Then Nil
        Else If time ≥ Tail(Tail(Head(evl)))
             Then Later(time, Tail(evl))
             Else Cons(Head(evl),Later(time,Tail(evl)));
    endwhererec; -- Quicksort

Get_Events
 = lambda (chs, n, t).
    If n < 1
    Then Nil
    Else If Ready(chs!n, t)
         Then Cons(Cons(n, event),
                   Get_Events (chs, n-1, t) )
         Else Get_Events (chs, n-1, t) ;
    where
    event = Cons(HeadCh(chs!n), Time(chs!n) ) ;
    endwhere ; -- Get_Events

New_Chan = lambda (ch, t).
           If Ready(ch, t)
           Then TailCh(ch)
           Else ch ;

endwhererec ; -- inputs
```

```
Monitor
= lambda (inputs, state, player_pos, score) .
   score_out
   whererec
   event      = Head(inputs) ;
   event_type = Head(event) ;
   event_data = Head(Tail(event)) ;
   event_time = Tail(Tail(event)) ;
   out_time   = event_time + comp_delay ;
   comp_delay = 50 ;

   score_out = If Lookup (new_state, new_player_pos) = mine
               Then Cons (Cons(-1, out_time), Nil)
               Else If new_score ≠ score
                    Then Cons (Cons(new_score,out_time), rest)
                    Else rest ; -- score_out

   rest = Monitor(Tail(inputs),new_state,new_player_pos,new_score) ;

   new_score = If event_type = 5
               Then If cell_value > 0
                    Then score + cell_value
                    Else score
               Else score
               where
               cell_value = Lookup (state, event_data) ;
               endwhere ; -- new_score

   new_player_pos = If event_type = 5
                    Then event_data
                    Else player_pos ;

   new_state = If event_type ≤ 4
               Then Update (state, event_type, event_data)
               Else state
               whererec
               Update = lambda (st, n, d) .
                         If n ≤ 1
                         Then Cons(d,Tail (st))
                         Else Cons(Head(st),
                                   Update (Tail(st), n-1, d)) ;
               endwhererec ; -- new_state

    Lookup = lambda (table, n) .
              If n ≤ 1
              Then Head(table)
              Else Lookup (Tail(table), n-1) ;

    endwhererec ; -- Monitor

 endwhererec ; -- Monitor_Process
```

```
Configuration Minesweep
Output score, f_cell_1, f_cell_2, f_cell_3, f_cell_4, f_val
Input  keyboard, rnd_1, rnd_2, rnd_3, rnd_4
Is
score = Monitor_Process (f_cell_1,f_cell_2,f_cell_3,f_cell_4,f_val);

f_cell_1 = Cell_Process (rnd_1) ;
f_cell_2 = Cell_Process (rnd_2) ;
f_cell_3 = Cell_Process (rnd_3) ;
f_cell_4 = Cell_Process (rnd_4) ;
f_val    = Validate_Process (keyboard) ;
end.
```

References

[Abramsky & Sykes 85]

A virtual machine for applicative multiprogramming
S. Abramsky and R. Sykes
Functional Programming Languages and Computer Architectures
Springer-Verlag LNCS 201, 1985

[Apt & Plotkin 81]

A cook's tour of countable nondeterminism
K.R. Apt and G. Plotkin
8th International Colloquium on Automata, Languages and Programming
Springer-Verlag LNCS 115.

[Apt & Olderog 83]

Proof rules and Transformations dealing with fairness
K.R. Apt and E-R. Olderog
Science of Computer Programming No. 3, pp 65-100. 1983

[Ashcroft & Wadge 76]

LUCID - A Formal System for Writing and proving programs
E.A. Ashcroft and W.W. Wadge
SIAM Journal of Computing Vol. 5, No. 3. Sept 1976

[Ashcroft & Wadge 77]

LUCID, a NonProcedural Language with Iteration
E.A. Ashcroft and W.W. Wadge
Communications of the ACM Vol. 20, No. 7. July 1977

[Ashcroft & Wadge 80]

Some common misconceptions about Lucid
E.A. Ashcroft and W.W. Wadge
ACM SIGPLAN Notices Vol. 15, No. 10. Oct 1980

[Backus 78]

Can Programming be Liberated from the von Neumann style ? A functional style and its Algebra of Programs
J. Backus
Communications of the ACM Vol. 21, No. 8. August 1978

[Baker 78]

List Processing in Real Time on a Serial Computer
H.G. Baker
Communications of the ACM Vol. 21 No. 4 April 1978

[Bergerand et. al. 85]

Outline of a Real-Time Data Flow Language
J.L. Bergarand, P. Caspi, D. Pilaud, N. Halbwachs and E. Pilaud
Proceedings IEEE Real-Time Systems Symposium
San Diego, California December 1985.

[Bergerand et. al. 86]

Automatic Control Systems Programming using a Real Time Declarative Language
J.L. Bergarand, P. Caspi, N. Halbwachs and J.A. Plaice
IFAC/IFIP Symposium on Software for Computer Control {SOCOCO 86}
Graz (Austria) May 1986

[Berry & Cosserat 84]

The ESTEREL Synchronous Programming Language and its Mathematical Semantics
G. Berry and L. Cosserat
Proceedings Seminar on Concurrency, Carnegie-Mellon University
Pittsburg 1984.
Springer-Verlag LNCS 197

[Berry et. al. 86]

Synchronous Programming of Reactive Systems
G. Berry, P. Couronne and G.Gonthier
Proceedings France-Japan Artificial Intelligence and Computer Science Symposium
Institute for New Generation Computer Technology, October 1986

[Berry et. al. 87a]

Synchronous Programming of Reactive Systems : An Inroduction to ESTEREL
G. Berry, P. Couronne and G.Gonthier
INRIA Rapports de Recherche No. 647. March 1987

[Berry et. al. 87b]

The ESTEREL v2.2 System Manuals
G. Berry, P. Couronne and G.Gonthier
Rapports Technique ENSPMP / INRIA May 1987

[Bird & Wadler 88]

Introduction to Functional Programming
R. Bird and P. Wadler
Prentice-Hall International 1988

[Bjorner & Oest 80]

Towards a Formal Description of Ada
Eds : D. Bjorner and O.N. Oest
Springer-Verlag LNCS 98

[Broy 82]

A Fixed point approach to Applicative Multiprogramming
M. Broy
Theoretical Foundations of Programming Methodology
Eds. M. Broy and G. Schmidt
D. Reidel Publishing Company 1982

[Broy 83]

Applicative Real-Time Programming
M. Broy
Proceedings IFIP 1983
North-Holland Information Processing 1983.

[Burstall & Darlington 77]

A Transformation System for Developing Recursive Programs
R.M. Burstall and J. Darlington
Journal of the ACM Vol. 24, No. 1. January 1977

[Burstall et. al. 80]

HOPE : An Experimental Applicative Language
R.M. Burstall, D.B. MacQueen and D.T. Sanella
Report CRS-62-80 May 1980
Dept. of Computer Science, University of Edinburgh.

[Burton 88]

Nondeterminism with referential transparency in Functional Programming Languages

F.W. Burton

The Computer Journal Vol. 31, No. 3. 1988

[Cartwright & Donahue 82]

The Semantics of Lazy {and Industrious} Evaluation

R. Cartwright and J. Donahue

Proceedings 1982 Symposium on Lisp and Functional Programming

Pittsburg 1982

[Caspi et. al. 87]

LUSTRE : A Declarative Language for Programming Synchronous Systems

P. Caspi, D. Pilaud, N. Halbwachs and J.A. Plaice

Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages

January 1987

[Darlington & Reeve 81]

Alice, a multiprocessor reduction machine for the parallel evaluation of applicative languages

J. Darlington and M. Reeve

Internal Report, Dept of Computing Science, Imperial College 1981.

[Darlington 82]

Program Transformation

J. Darlington

Functional Programming and its Applications

Eds. J. Darlington, P. Henderson and D.A. Turner

Cambridge University Press 1982

[Faustini & Lewis 85]

A Declarative Language for the Specification of Real Time Systems

A.A. Faustini and E.B. Lewis

Proceedings IEEE Real-Time Systems Symposium

San Diego, California. December 1985.

[Faustini & Lewis 86]

Toward a Real-Time Dataflow Language
A.A. Faustini and E.B. Lewis
IEEE Software January 1986

[Friedman & Wise 76]

Cons should not evaluate its arguments
D.P. Friedman and D.S. Wise
Automata Languages and Programming Third International Colloquium
Eds. S. Michaelson and R. Milner
Edinburgh University Press 1976

[Harrison 87]

RUTH : A Functional Language for Real-Time Programming
D. Harrison
Proceedings Parallel Architectures and Languages Europe
Eindhoven, June 1987
Springer-Verlag LNCS 259

[Harrison & Nelson 89]

A Real-Time Language Based on Explicit Timeouts
D. Harrison an S. Nelson
University of Stirling, Department of Computing Science
Technical Report 1989

[Henderson & Morris 76]

A Lazy Evaluator
P. Henderson and J.H. Morris
Conference Record of the 3rd Annual ACM symposium on Principles of Programming Languages
January 1976

[Henderson 80]

Functional Programming : Application and Implementation
P. Henderson
Prentice-Hall International 1980

[Henderson 82]

Purely Functional Operating Systems
P. Henderson
Functional Programming and its Applications
Eds. J. Darlington, P. Henderson and D.A. Turner
Cambridge University Press 1982

[Henderson et. al. 83]

The LispKit Manual
P. Henderson, G.A. Jones and S.B. Jones
Oxford University Computing Laboratory
Programming Research Group Monograph 32

[Holmstrom 83]

PFL : A Functonal Language for Parallel Programming
S. Holmstrom
Proceedings Declarative Programming Workshop
University College London 11-13th April 1983

[Hudak et.al. 89]

Report on the Functional Programming Language Haskell
P. Hudak (Editor), P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, J. Hughes, T. Johnsson, D. Kieburtz, S. Peyton Jones, R. Nikhil, M. Reeve, D. Wise and J. Young
University of Glasgow, Department of Computing Science,
Research Report CSC/89/R5 March 1989

[Ida & Tanaka 83]

Functional Programming with Streams
T. Ida and J. Tanaka
Proceedings IFIP 1983
North-Holland Information Processing 1983

[Ida & Tanaka 84]

Functional Programming with Streams - Part II
T. Ida and J. Tanaka
New Generation Computing 2, pp 261-275. 1984

[INMOS 84]

OCCAM Programming Manual
Inmos Ltd.
Prentice-Hall International Series in Computer Science 1984

[INMOS 87]

Preliminary Data : IMS T414 Transputer
Inmos Ltd. 1987

[Jones 84a]

Abstract Machine Support for Purely Functional Operating Systems
S.B. Jones
University of Stirling, Department of Computing Science
Technical Report T.R.15. September 1984

[Jones 84b]

A Range of Operating Systems Written In A Purely Functional Style
S.B. Jones
University of Stirling, Department of Computing Science
Technical Report T.R.16. September 1984

[Jones & Sinclair 89]

Functional Programming and Operating Systems
S.B. Jones and A.F. Sinclair
The Computer Journal Vol. 32, No. 2. Feb. 1989

[Lamport 78]

Time, Clocks and the Ordering of Events in a Distributed System
L. Lamport
Communications of the ACM Vol. 21, No. 7. July 1978

[Landin 64]

The Mechanical evaluation of Expressions
P.J. Landin
The Computer Journal Vol. 6 pp308-320. 1964

[Le Guernic et. al. 85]

SIGNAL : A Data Flow Oriented Language for Signal Processing
P. Le Guernic, A. Benveniste, P. Bournai and T. Gautier
IRISA Internal publication 246. January 1985

[Le Guernic et. al. 86]

SIGNAL - A Data Flow-Oriented Language for Signal Processing
P. Le Guernic, A. Benveniste, P. Bournai and T. Gautier
IEEE Transactions on Acoustics, Speech and Signal Processing Vol. ASSP-34, No. 2. April 1986

[Le Guernic & Benveniste 87]

Real-Time, Synchronous, Data-Flow Programming : The Language SIGNAL and its Mathematical Semantics

P. Le Guernic and A. Benveniste

INRIA Rapports de Recherche No. 620. February 1987

[Le Lann 83]

On Real-Time Distributed Computing

G. Le Lann

Proceedings IFIP 1983

North-Holland Information Processing 83.

[Liebermann & Hewitt 83]

A Real-Time Garbage Collector Based on the Lifetimes of Objects

H. Liebermann and C. Hewitt

Communications of the ACM Vol. 26 No. 6. June 1983

[MoD 70]

Official Definition of Coral-66

Ministry of Defence 1970

[McCarthy 60]

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

J. McCarthy

Communications of the ACM Vol. 3, No. 4. April 1960

[McCarthy 63]

A basis for a mathematical theory of computation

J. McCarthy

Studies in logic: Computer programming and formal systems

Eds. Braffort and Hirschberg

North Holland 1963.

[Partsch & Steinbruggen 83]

Program Transformation Systems

H. Partsch and R. Steinbruggen

ACM Computing Surveys Vol. 15, No. 3. September 1983

[Peyton Jones 85]

GRIP - a parallel graph reduction machine
S.L. Peyton Jones
University College London Internal note 1665. January 1985

[Peyton Jones 87]

The Implementation of Functional Programming Languages
S.L. Peyton Jones
Prentice-Hall International 1987

[Sannella 81]

HOPE Update
D. Sannella
University of Edinburgh, Dept. of Computer Science. February 1981

[Schmidt 86]

Denotational Semantics, A Methodology For Language Development
D.A. Schmidt
Allyn and Bacon 1986

[Sherlis 80]

Expression Procedures and Program Derivation
W.L. Sherlis
Ph.D. Thesis
Dept. Of Computer Science, Stanford University
Report No. STAN-CS-80-818. August 1980

[Shin 87]

Introduction to special issue on Real-Time systems
K.G. Shin
IEEE Transactions on Computers Vol. c-36, No. 8. August 1987

[Stoy 77]

Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory
J. Stoy
The MIT Press 1977

[Stoye 86]

Message-Based Functional Operating Systems
W. Stoye
Science of Computer Programming No. 6, pp 291-311. 1986

[Thompson 86]

Writing Interactive Programs in Miranda
S. Thompson
UKC Computer Lab. Report No. 40. University of Kent, August 1986

[Turner 85]

Miranda : A non-strict functional language with polymorphic types
D.A. Turner
Proceedings, Functional Programming Languages and Computer Architecture
Nancy, France. Sept 1985
Springer-Verlag LNCS 201

[Turner 87]

Functional programming and communicating processes
D.A. Turner
Proceedings Parallel Architectures and Languages Europe
Eindhoven. June 1987
Springer-Verlag LNCS 259

[USDOD 83]

Programming Language Ada : Reference Manual
U.S. Dept. of Defence 1983
Springer-Verlag LNCS 155

[Wadge & Ashcroft 85]

Lucid, the Dataflow Programming Language
W.W.Wadge and E.A.Ashcroft
Academic Press, London 1985

[Wirth 83]

Programming in Modula-2
N. Wirth
Springer-Verlag 1983

[Young 82]

Real Time Languages design and development
S.J. Young
Ellis-Harwood Publishers 1982.