A semantics for functions and behaviours

Anthony Charles Daniels

Thesis submitted to The University of Nottingham for the degree of Doctor of Philosophy
December 1999

Abstract

The functional animation language Fran allows animations to be programmed in a novel way. Fran provides an abstract datatype of "behaviours" that represent time varying values such as the position of moving objects, together with a simple set of operators for constructing behaviours. More generally, this approach has potential for other kinds of real-time systems that consist of interactive components that evolve over time.

We introduce a small functional language, CONTROL, which has behaviours and operators that are similar to those in Fran. Our language improves on Fran in certain key areas, in particular, by eliminating start times and distinguishing between recursive functions and recursive behaviours. Our main contribution is to provide a complete formal semantics for CONTROL, which Fran lacks. This semantics provides a precise description of the language and can be used as the basis for proving that programs are correct.

The semantics is defined under the assumption that real number computations and operations on behaviours are exact. Behaviours are modelled as functions of continuous time, and this approach is combined with the standard approach to the semantics of functional languages. This combination requires some novel techniques, particularly for handling recursively defined behaviours.

Contents

List of Figures

Acknowledgements

I would particularly like to thank Conal Elliott for introducing me to this absorbing topic, guiding me through the early stages of my PhD., and persuading the Microsoft Corporation to fund this work. Next I would like to thank Mark P. Jones for his conscientious supervision and inspirational tutoring at Nottingham. I thoroughly enjoyed the many hours of engaging discussion we had on this topic. Graham Hutton took over supervision during the writing up stage, and I greatly appreciate his useful feedback and advice in general. Paul Blampied gave some detailed comments on early chapters which improved the presentation. Thanks also to all the other members of the Languages and Programming group for creating such a stimulating environment, and particularly to Colin, Ben, Claus and Paul for many interesting discussions.

Outside work my sanity has been revived by friends in Nottingham and elsewhere. Particular thanks to Jenny for her constant encouragement and expert assistance with grammatical aspects of my writing. For less sober assistance I thank Colin, Felix, Jason, Vicky, Rosey, Bob, Bill, Ben, Katie, Al, Chris and Alex. You made my time at Nottingham an unforgettable experience.

Most of all I would like to thank my parents for their support and encouragement in everything I've ever done, and finally my Gran for buying Rosey and me our first computer.

Notation

$x = y$	x and y are semantically equal
$x \equiv y$	x and y are syntactically identical
$\mathbb{R}$	set of real numbers
$\mathbb{B}$	set of boolean values, $\{true, false\}$
$\mathbb{T}$	set of times (non-negative real numbers, $\{t \in \mathbb{R} \mid t \geq 0\}$)
$A \to B$	functions from A to B (when A and B are sets)
$A \rightharpoonup B$	partial functions from A to B
$t \mapsto X$	function mapping the bound variable t to the formula X
$\left\{ \begin{array}{c} X_1 \\ X_2 \end{array} \right| C_1$	conditional function $\begin{cases} X_1 & \text{, if } C_1 \\ X_2 & \text{, otherwise} \end{cases}$
$D \to E$	ω-continuous functions from D to E (when D and E are domains)
$\bot_D$	bottom element of the domain D
$X_\bot$	flat domain formed by lifting the set X
$[\![T]\!]$	domain corresponding to type T
$[\![E]\!]$	semantic function interpreting term E

$\emptyset$	empty set, $\{\}$
$\mathbb{P}(A)$	power set of A, $\{S \mid S \subseteq A\}$
$X \setminus Y$	set difference, $\{x \in X \mid x \notin Y\}$
$X \supsetneq Y$	proper superset, $X \supseteq Y \wedge X \neq Y$
$\uparrow S$	upperset of S, $\{s \in \mathbb{R} \mid \exists s' \in S : s' \leq s.\}$
$[a, b]$	closed interval, $\{x \in \mathbb{R} \mid a \leq x \leq b\}$
(a, b)	open interval, $\{x \in \mathbb{R} \mid a < x < b\}$
$[u \mid a : x]$	function identical to u except that a maps to x
$C[\_]$	context
E/δ	substitution of all free variables in E by δ
$E/[x : N]$	substitution of x for N in E
$\Gamma \vdash E : \theta$	typing judgement that E has type θ in context Γ
$FV(E)$	set of free variables of the term E
$\mathrm{dom}\ f$	domain of the function f
$\to$	evaluation relation
$\longrightarrow$	transition system
ε	empty term
$\bigsqcup_{n=0}^{\infty} X_n$	least upper bound of $\{X_0, X_1, \dots\}$

Chapter 1

Introduction

> Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things.
>
> P. J. Landin

In 1966 Landin proposed a core language framework based on the λ-calculus which he called ISWIM [Lan66]. His hypothesis was that this framework could provide a basis for many realistic languages, each one differing only in the set of given things that are required for programming particular kinds of applications. This is a natural approach to take because it is clear that some tasks are easier in certain languages than others, but also that there are similar core features in most languages; for example, most languages provide facilities for defining functions and values, controlling the scope of identifiers, expressing conditionals and building data structures. Landin went on to suggest:

> A possible first step in the research programme is 1700 doctoral theses called "A correspondence between x and Church's λ-notation."

We follow Landin's approach by proposing a new core language called CONTROL (CONtinuous Time Reactive Object Language) which is intended as a basis for reactive systems languages. Reactive systems control and monitor various entities in real-time, and CONTROL provides operations for describing time-varying quantities using *behaviours*. In common with ISWIM, CONTROL is based on the λ-calculus, but it uses types and normal order evaluation so it is actually more closely related to PCF [Sco93].

Like Landin, we also want to give an unambiguous description of our language. In the three decades since Landin's paper, much progress has been made towards constructing theories of programming languages [Rey98]. These theories allow us to rigorously define the meaning of programs written in a particular language; in other words, they allow us to construct a semantics for the language. The functional part of CONTROL can be dealt with using these existing theories, but behaviours require some new techniques; in particular, for describing reactivity, integrals and for defining behaviours recursively. The development of these techniques, and of a complete semantics for CONTROL, are the primary subjects of this dissertation.

1.1 Reactive systems

In our context, reactive systems encompass any real-time control system that must respond to external stimuli in non-trivial ways; for example, lifts, robots, aircraft, heating systems, power stations, satellites and interactive animations. Although these systems are physically very dissimilar (particularly interactive animation) descriptions of their abstract behaviour often bear a strong resemblance. What all these systems have in common is that they must monitor values that vary with time and respond or react to situations that arise; we call these situations *events*. Furthermore, events cannot in general be predicted prior to the system being run, and so the responses must be determined in real-time. This is in contrast to simpler real-time systems, such as signal processing, where the computation that will be performed is fixed and not determined by occurrences of events.

One of the most difficult problems reactive systems pose is that computing responses to events takes a finite amount of time, and this results in a delay in the response. Of course, in any physical system there will always be a slight delay due to the mechanical or electrical hardware, but this is generally a fixed period as opposed to the variable time taken to compute responses. Events cannot usually be predicted beforehand, and it may be possible for many events to occur in a short period of time. If the responses to these events cannot be computed quickly enough, then the system could fail.

Reactive languages are designed to make it easier to program reactive systems. We discuss various reactive languages in Chapter 2. One of the major tasks in specifying reactive systems is defining when response times are acceptable. For this purpose various calculi have been devised. They help system designers to reason about real-time properties of programs, and have been used to verify that implementations of particular systems meet their specification. Despite these formalisms, it is usually very difficult to guarantee real-time properties of real systems, largely because the languages used are not amenable to analysis using real-time calculi.

1.2 Fran

CONTROL is inspired by the animation language Fran which takes a different approach to that of most reactive languages. It adopts a continuous notion of time so that, as far as the programmer is concerned, responses occur at exactly the time specified and time-varying quantities (i.e., behaviours) vary continuously rather than being approximated by values at discrete points in time. Fran provides an abstract datatype for behaviours embedded in an existing functional language, Haskell.

In contrast with Fran, most other reactive languages are based on discrete time. Fran does not make any guarantees about response times, however, because the implementation of the language features uses discrete representations, and computation speed is, of course, finite. But the shift of emphasis is important: if it can be shown that the language features satisfy certain real-time constraints for all programs, subject to some limiting criteria, then the task of verifying the real-time properties of individual programs is greatly simplified. So the burden of proof is shifted from individual programs onto the language, which potentially saves much repetition of effort.

This was not the original motivation for Fran; it was intended to reduce the repetition of effort in programming computers to display modelled animations. We discuss Fran in detail and expand on this point in Chapter 3. Our work uses continuous time as Fran does, but for the benefits it brings to proving programs are correct as well as for ease of programming.

1.3 Approach

> It is true that our machines can only provide an approximation to these [real valued] functions but the discrepancies are generally small and we usually start by ignoring them. It is only after we have devised a program which would be correct if the functions used were the exact mathematical ones that we start investigating the errors caused by the finite nature of our computer.
>
> C. Strachey

CONTROL has a type for real numbers along with various operations on this type. In addition, behaviours can be real-valued and they can be integrated and used to describe events. As Strachey observed[Str73], when considering the semantics of such operations on real numbers we must first consider the outcome if the functions were the exact mathematical ones, and then consider the errors due to approximation. This was before methods had been devised for computing certain operations on exact representations of real numbers [Vui90]. However, Strachey's approach is still applicable to CONTROL because the operations available in the language are beyond the limits of these methods, and therefore most implementations of CONTROL are likely to use approximation techniques for these operations.

This work accomplishes the first step suggested by Strachey—finding the values assuming the operations are exact—and does not address the second step—investigating the errors in an implementation using approximation techniques. It is therefore an idealised theory which provides a theoretical basis for the study of real languages based on CONTROL. Without this theory we would not know what it is that these real languages are approximating.

1.4 Assumptions

Our aim is to construct a formal semantics for an idealised language based on the the core operators in Fran. There is some flexibility here because the actual language is not specified completely. It includes the most important features in Fran, but we may alter the language design in response to semantic considerations. In other words, we will describe the design of a Fran-like language, rather than a language that is a strict subset of Fran.

There are a number of assumptions we make; firstly, the validity of Strachey's approach of starting with an idealised language and then considering separately the errors due to approximation. From a programming perspective, we assume that continuous time behaviours and declarative programming are sufficient, and also convenient, for creating many reactive systems. That said, we feel that the example applications in Chapter 9 demonstrate the power and elegance of CONTROL for simple reactive systems.

1.5 Contributions

Our contributions are twofold: firstly, we have designed a new language which improves on previous languages; and secondly, we have constructed a formal semantics for our language which has not been accomplished for similar languages. It is likely that our semantics could be adapted to account for features in related languages such as Fran.

More specifically, our contributions towards language design are: an implicit notion of time that makes explicit time values unnecessary; a new mechanism for defining recursive behaviours; and the integration of these features into a purely functional language. Our main contributions towards semantics are: a formal definition of the core operators (Chapter 5), in particular, a more refined treatment of event occurrences; a semantics for recursive behaviour definitions (Chapter 6); a complete semantics, combining functions and behaviours (Chapter 4, Chapter 7, Chapter 8); and some useful theorems for proving properties of programs (Chapter 8). Finally, we also give some examples which illustrate the expressiveness of the language and the usefulness of the semantics (Chapter 9).

1.6 Advice to the reader

The remainder of this dissertation is organised as follows. Chapter 2 gives some background on various reactive languages and Chapter 3 describes Fran in detail. The subsequent chapters describe our contribution—a language for programming reactive systems with continuous time behaviours and its formal semantics. The quickest way to find the main technical results is to read Chapter 8 which gives the complete formal semantics of CONTROL. Doing so would miss the motivation behind the language features and the semantics, but it would reveal the flavour of this work. The examples in Chapter 9 are also worthwhile for those not wanting to read the dissertation in full.

For the most part, chapters begin by following the main development of the theory with few deviations. Then, towards the end of each chapter, various interesting alternatives and additional parts of theory are explored in sections titled 'Avenue on x.' This makes it possible to concentrate on the main discussion uninterrupted, and then explore other possibilities separately. In fact, one useful way to approach this dissertation is to first read it through ignoring the avenue sections, and then re-read it in full.

Chapter summary

CONTROL is a functional language with facilities for describing time-varying quantities called behaviours, where time is continuous. Our theory of the meaning of programs is idealised because it assumes that all real number computation, including integration and comparison of behaviours, is exact. CONTROL evolved from a core subset of Fran, which is a functional language for animation. This dissertation develops a complete formal semantics for

CONTROL and illustrates the application of this theory.

Chapter 2

Background

This chapter begins with a survey of some languages for programming reactive systems. The aim here is to concentrate on the features in each language and to give an idea of the character of programs. The languages we discuss at first are based on a discrete model of time, in contrast to our language. Then we discuss the relative merits of continuous time compared to discrete time, and discuss a language based on continuous time.

Following this we review two calculi for specifying real-time properties of programs. The first is the Duration Calculus, which we illustrate with the standard gas burner example. The second is an extension of CSP for specifying hybrid systems. We describe a water tank controller using this notation, and in Chapter 9 we will return to this example and program it in CONTROL.

2.1 Esterel

Esterel is an imperative reactive language designed for programming control intensive reactive systems [Ber97]. Here 'reactive' means that the role of the system is to react to external stimuli in a timely way; the pace of the interaction is determined by the environment. Esterel is deterministic, which means that the output of the system is uniquely determined by the inputs and their timing. Esterel is based on a synchronous model of concurrency; that is, one in which concurrent processes are able to perform computation and exchange information in zero time. In practice, however, it is not possible to implement such a model exactly, so timing constraints are based on estimates and are not guaranteed.

As an example of concurrency in Esterel, consider the following:

$$\texttt{await } A \texttt{ || await } B.$$

This is a process which terminates as soon as the input actions A and B have both occurred. Processes can also be pre-empted, which means interrupted by another process that has priority; for example, the process

$$\texttt{loop } P \texttt{ each } R$$

acts like the process P until the event R occurs, at which point P is started afresh. Putting these two examples together, and adding an output action `emit O`, we obtain the following Esterel code fragment which emits the output 0 as soon as both the inputs A and B have been received, and resets whenever the input R is received;

```
loop
    [await A || await B];
    emit O
each R.
```

Because Esterel performs actions and all communication instantaneously, events can occur simultaneously and pre-emption is instant. In practice, the usual way that implementations work is to deal with all active processes in each input-output cycle. For example, in the program above we would first receive any input from A, B, and R, and then determine what action to take. Since R is a pre-emption process it takes priority over A and B, so any input to R will restart the program. After all inputs have been received, all outputs are sent and a new input-output cycle begins. Therefore the discrete notion of time is essential for achieving synchronicity.

2.2 Lustre

Lustre is a dataflow programming language designed for programming reactive systems [HCRP91]. It is well suited for data intensive reactive systems, such as signal processing, in contrast to Esterel which is aimed at control intensive systems. In common with Esterel, it uses the synchronous model of communication. Verification of timing properties is an important concern, and the designers of Lustre claim that this is simpler than for some other languages because of its similarity with temporal logics. This allows the language to be used for both writing programs and expressing program properties, easing the task of verification. Later on we will see that it is possible to apply this idea to CONTROL programs.

Lustre models time-varying quantities by sequences of values, called flows, and so it is based on a discrete notion of time. We will use integer indices to refer to the value of a flow at a discrete point in time. Lustre flows are quite similar to difference equations, which are equations in terms of sequences (Hubbard and West describe difference equations as evolution in discrete time, compared to differential equations which describe evolution in continuous time [HW91]). One difference, however, is that the operations provided in Lustre are deliberately restricted so that flows are straightforward to compute.

As a simple example, we will describe how to represent the sequence of Fibonaci numbers by a flow in Lustre. The Fibonaci sequence is usually defined by the following equations:

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2}. \end{aligned}$$

The third equation states that the n-th Fibonaci number is the sum of the previous two numbers in the sequence. Therefore we need to know how to refer to earlier values in flows, and how to add them.

To refer to earlier values in a flow the `pre` operation is used. It offsets flows so that, for example,

$$Y\ \texttt{=}\ \texttt{pre(}X\texttt{)}$$

defines Y to be the flow that has values equal to the previous values of X, that is,

$$Y_t = X_{t-1}.$$

The first value, Y_0, of the stream Y is uninitialised, but it can be set using the `->` operator. Thus the flow Z given by

$$Z\ \texttt{=}\ \texttt{2.->pre(}X\texttt{)}$$

is identical to Y except that the first value is 2 rather than being uninitialised; that is, $Z_0 = 2$.

Operations are defined element-wise, so, for example, addition of flows satisfies

$$X\ \texttt{+}\ Y = \{X_t + Y_t \mid t \in \mathbb{N}\}.$$

Now, using `pre`, `->` and `+`, we can define the flow of Fibonaci numbers by

```
F = 1.->1.->(pre(F) + pre(pre(F))).
```

Reactivity is expressed by forming boolean valued flows. This bears a close resemblance to imperative streams which we will describe in the next section. Although Lustre is intended to be amenable to program verification, its discrete model of time makes this awkward for many applications that are most naturally specified using continuous time values.

The language Signal [LGLL91] is similar to Lustre, and is also based on flows.

2.3 Imperative streams

Streams view input devices as sequences of values and produce corresponding streams of output values [KM77]. Imperative streams are a generalisation which allow side effects with each value in the stream [Sch96b]. They have been implemented as a monad called `ST` in Haskell; a value of type `ST a` represents an imperative program which produces values of type a at certain times during its execution. Using a monad for imperative streams has the advantage that arbitrary IO commands can be performed as in the IO monad. The difference is that a value of type `IO a` represents an imperative program that will produce a single value of type `a` at the end of its execution, rather than a stream of values. Imperative streams can be used to model change over time and handle streams of input values, so they are suitable for programming reactive systems. They have been used for the graphical user interface toolkit PIDGETS [Sch96a, Sch98].

Imperative streams yield values at certain times, so they are a discrete representation of time-varying values. Streams may need to wait for values from other streams before yielding a new value, so timing constraints are implicit and real-time response is not guaranteed.

An `until` operator similar to `until` in Arctic and `untilB` in Fran is discussed in [Sch96b]; the stream `until c b d` behaves like `b` until `c` produces `True` and then it behaves like `d`. Scholz compares imperative streams with Fran in his thesis [Sch98]. The monadic, discrete approach of imperative streams results in a more imperative, state based style of programming compared to Fran's purely declarative style. Finally, Sage has established an even closer link by re-implementing Fran using imperative streams [Sag98].

2.4 Real-time process calculi

Many reactive languages make use of concurrency where multiple processes run in parallel. Concurrency is useful for programming reactive systems because they involve a number of entities that interact with each other. The software must monitor and control these entities and so multiple interacting processes provided for by concurrency is very natural.

There are two main forms of concurrency: shared variable concurrency, where certain variables are shared by multiple processes [Hoa72]; and communicating sequential processes (CSP), where processes communicate by passing messages [Hoa85]. The main difference is in the way processes communicate with each other.

Another distinction is between synchronous and asynchronous languages; in synchronous languages the input and output of a message occur simultaneously, and processes use what is known as handshake communication. In asynchronous languages the sender does not need to wait until the receiver is ready, so there is no handshake.

A common alternative to passing messages between processes is to use named channels so that a process can request an input v from a channel h, written $h?v$, or output a value e along a channel h, written $h!e$. This approach is captured algebraically by the π-calculus [Mil91].

CSP does not provide facilities for synchronising with a clock. This is addressed in a variation called Timed CSP which allows explicit reference to timing information [RR87, Sch90]. However, it is often not necessary to use concurrency explicitly to describe reactive systems; for example, languages such as Fran use a declarative style where values are defined in terms of each other, and the processes for computing these values are built into the language. It may be beneficial to use concurrency for implementing declarative reactive languages such as Fran, but it is not necessary to provide concurrency in the language.

2.5 Continuous verses discrete time

There is a fundamental conceptual difference between discrete and continuous views of time. The exact nature of space and time has intrigued philosophers for centuries, and many metaphysical arguments have been put forward in support of each viewpoint. The outcome of these arguments depends on the assumptions they are based on, so they do not provide a conclusive answer. More recently, advances in Physics have changed our perspective, suggesting that space and time are non-linear and also that both may be discrete. For our purposes all this is not very important. In most situations time appears to flow continuously, and we do not perceive an uneven progression from one instant to another. If space and time are discrete then the granularity is so fine that for all practical purposes they appear to be continuous. This is why continuous time models have been so successful in science and engineering. A good model is one that is workable and fits observation well, and not necessarily one that mirrors reality most accurately.

Continuous time models have the following benefits. They yield a value at any point in time, not just discrete points. They are easier to manipulate in symbolic form, and have many algebraic properties. Most importantly, they can be integrated and differentiated. Calculus is perhaps the most powerful tool ever developed for scientists and engineers.

In practice, however, continuous time is less common than discrete time. We have seen that Esterel, Lustre and Imperative streams use discrete time representations for time-varying quantities. This is the most common approach because digital computers are discrete machines. It is possible to represent continuous time values using functions of time, but operations such as integration tend to be more difficult than for discrete time representations for which approximation methods can be used. It is possible to combine the approaches, however, and use discrete approximation methods for operations such as integration and interpolate between points to obtain continuous time values. This allows us to retain the continuous time approach, but we have to sacrifice exact operations where necessary. This is how Fran is implemented.

So, practically we usually need to compromise the continuous time approach by using discrete approximation methods for some operations. Although we have the extra complexity of analysing errors due to approximation, many of the advantages of using continuous time remain.

2.6 Arctic

Arctic [Dan84] is intended for implementing real-time control systems of the kind we have called reactive systems; that is, systems that require complex decision making and must satisfy hard timing constraints. The first criterion excludes signal processing, for example, and the second excludes soft real-time systems such as operating systems, where the user may have to wait for the machine at times.

Conceptually Arctic is based on a continuous notion of time, and inputs and outputs are modelled as functions of time. The implementation, on the other hand, uses discrete time and computes approximations to time-varying values. Arctic's creator, Dannenberg, claims that any Arctic implementation will only approximate the ideal because it is impossible to measure or represent real input values or times with infinite precision. While measurements in the real world are always approximations, it is not true that real numbers are impossible to represent exactly—for example [PEE97, Vui90]—although it can be very difficult to implement some operations on exact representations of real numbers. In summary, Arctic adopts continuous time conceptually—for understanding the language and for reasoning about programs—but does not make any guarantees about the implementation, and therefore program verification is compromised.

The principal construct in Arctic is the prototype, which is a specification of responses to events. Prototypes are instantiated to yield actual values (outputs) when they are triggered by events. Thus a single prototype may be instantiated many times by different events. Each of the resulting objects have their own state, which includes their start time.

As an example of a prototype, the following describes a doorbell that does not ring between 0am and 8am:

```
Push causes [
    if (time mod 24 hours) > 8 hours then RingBell].
```

Each `Push` event creates an instance of the prototype, which in turn will instantiate the `RingBell` output event if the time of the `Push` event satisfies the condition. The time of the event is given by `time` in the prototype—it is an implicit parameter which gives the start time of instances of the prototype.

Another implicit parameter that is passed to instances of prototypes is a stretch factor called `dur`. This allows a prototype to be speeded up or slowed down by adjusting all timed outputs according to the stretch factor. This is significant to our work because time-transformations such as speeding up timed values can be problematic with discrete time representations.

In the following program the `Ring3Times` event triggers a bell that rings three times at one second intervals:

```
Ring3Times causes [
    RingBell @ 0
    RingBell @ 1
    RingBell @ 2].
```

Here the operator `@` is used to specify the time of output events relative to the start time of the prototype instance. Thus, if the bell is pressed at time t then we expect the bell to ring at times t, t+1 and t+2. Now, we can double the speed of this prototype using the $\sim$ operator so that the bell rings at times t, t+0.5 and t+1. So time stretching affects the relative times but not the start time, which means that the `@` operator multiplies the given time by the stretch factor, `dur`, to obtain the actual time. This ensures time-transformations interact with composition of prototypes as we would expect.

Arctic includes primitives for parallel and sequential composition of prototypes. Sequential composition uses a third implicit parameter called `stop` which gives the end time of the prototype. There are facilities for constructing functions of time and for describing events. In particular, the `until` operator evaluates a boolean function of time and switches from one prototype to another at the first moment after the start time when the function yields true. This is ill defined, however, because a boolean function such as $t \mapsto t > 1$ does not have a first moment when it is true. Later we will see how our theory avoids this flaw in the definition of a similar operator in our language.

2.7 Duration Calculus

The Duration Calculus is for reasoning about specifications and designs of real-time systems. It is closely related to interval temporal logics, which allow assertions about timing to be specified without explicit references to absolute time [Koy90, MP92]. However, it also has a simple way of describing the proportion of time a system spends in a given state, which can be useful in some applications. Real numbers are used to model time and boolean valued functions of time are used to model states and events. This gives a continuous notion of time.

The standard example is a gas burner, which can only leak unlit gas for one twentieth of the elapsed time without a dangerous build up occurring. A dangerous buildup cannot occur in less than one minute. Let

$$Leak(t) : Time \rightarrow \mathbb{R}$$

be a real valued function of time that is 1 when the gas burner is leaking and 0 when it is not. The safety requirement over the interval $[b, e]$ is given by,

$$e - b \geq 60sec. \implies \int_b^e Leak(t).dt \leq \frac{1}{20}(e - b).$$

So the total length of time when the burner is leaking is found by integrating $Leak$, and this should be no more than one twentieth of the total time.

This specification can be simplified by eliminating explicit references to time. We assume that all integrals are over the interval $[b, e]$, so that the elapsed time, l, is given by:

$$l = \int 1 \qquad (= \int_b^e 1.dt = e - b).$$

Now the safety requirement can be expressed as:

$$l \geq 60sec. \implies \int Leak \leq \frac{1}{20}l.$$

The Duration Calculus is defined axiomatically in terms of integrals such as the one above and arbitrary states. It has been used successfully to specify a variety of reactive systems and to prove designs are correct.

2.8 Hybrid systems

Hybrid systems combine continuous devices and discrete control programs, typically in real-time systems where the physical environment is evolving over time. The combination of continuous and discrete values makes it challenging to specify and implement provably correct systems.

We have already seen the Duration Calculus for specifying and reasoning about discrete states in real-time systems. This has been extended to capture piecewise continuous states so that it can be used for specifying hybrid systems [CRH93]. The theory is intended to interface with mathematical analysis which is required for analysing the continuous parts of the system.

He Jifeng has described hybrid systems using an extension of CSP with a specification oriented semantics [Jif94]. We will briefly consider an example of describing a hybrid system using this method. This system controls the water level in a tank by switching on a control valve. The level must be kept between 30 and 60 units, and starts at time 0 at 40 units. The valve is open which causes the water level to rise at 0.2 units per second. Once the valve is closed the level will drop at 0.1 units per second until the valve is reopened. We have the following variables:

WL	is the hybrid system,
h	is the water level,
C	is the controller,
c	is the channel that links the controller with the valve.

The system is specified in He's notation by:

$$
\begin{aligned}
WL &\stackrel{def}{=} (\dot{h} = 0.2)_{40} \trianglelefteq (c?x \longrightarrow W(x)) \\
W(\text{off}) &\stackrel{def}{=} (\dot{h} = -0.1) \trianglelefteq (c?x \longrightarrow W(x)) \\
W(\text{on}) &\stackrel{def}{=} (\dot{h} = 0.2) \trianglelefteq (c?x \longrightarrow W(x)) \\
C &\stackrel{def}{=} (\text{await } h = 30 \text{ do } (c!\text{on} \longrightarrow (\text{delay } 1; C))) \oplus \\
&\phantom{\stackrel{def}{=}} (\text{await } h = 60 \text{ do } (c!\text{off} \longrightarrow (\text{delay } 1; C)))
\end{aligned}
$$

The controller C opens the valve when the water level drops to 30 units and closes it when the water level rises to 60 units. Opening and closing is achieved by passing the values 'on' or 'off' along the channel c. The definitions for W specify the rate of level change, $\dot{h}$, when the valve is off (closed) and on (open), and the change that occurs when an input is received on channel c according to W. The overall water level system, WL, gives the rate of level change, $\dot{h}$, which is 0.2 at first, with $h = 40$, and changes when an input is received on c.

This notation is precise and amenable to reasoning and proving programs correct. However, in Chapter 9 we show how this system can be implemented in CONTROL, and that our notation is precise enough for both the specification and the implementation. This has the advantage that the program is correct by design.

Chapter summary

There are many languages designed for programming reactive systems using a variety of techniques. We saw that most of these languages are based on a discrete notion of time. One exception is Arctic which adopts continuous time conceptually.

Specification calculi often assume that time is continuous, because it is easier to work with, but it is then not easy to prove that an implementation is correct with respect to its (continuous time) specification unless it is written in a language that supports continuous time.

Chapter 3

The Fran system

Fran (Functional Reactive ANimation) is a functional language for creating interactive animations. CONTROL is based on a core fragment of Fran with the intention of studying the semantics of the core operators in a simpler functional language. In retrospect, it is apparent that our work has wider significance to reactive languages, and this is what we have emphasised in the previous chapters. Because of its influence on our work, we will describe Fran, and the existing work on its semantics, in detail in this chapter. The main purposes are:

- To introduce Fran's operators on behaviours, which inspired similar operators in CONTROL.
- To enable us to identify where CONTROL differs from Fran.
- To describe previous work on the semantics of Fran, and its limitations.

Fran is the latest prototype language in a research programme investigating high-level languages for creating richly interactive animations. The ideas that Fran is based on grew out of earlier work by Elliott and others on modelled animation [ESYAE94, Ell96]. Fran is implemented in Haskell, but most of the ideas behind the design are independent of the implementation language. Therefore it is helpful to distinguish between the key concepts behind the approach and specific details of the Haskell based implementation; we are mostly interested in the former, but our example programs will use the Haskell implementation of Fran.

3.1 Examples

In this section we shall describe a simple Fran animation to illustrate some of the operators in the language and to give an impression of Fran so that the discussion of the key concepts which follows is more concrete.

Our first example is an animation of the Moon orbiting the Earth in a circular path. We require a time-varying value (a behaviour) which gives the position of the Moon at all times. Then we can translate an image of the Moon according to this behaviour, and overlay it on a stationary image of the Earth. By default, images that are not translated, such as the Earth in this example, are positioned at the origin which is at the centre of the display. The Fran program for this animation is as follows:

```
orbit = vector2XY (cos time) (sin time)
earthMoon = move orbit moon ‘over‘ earth        (3.1)
```

The definition of `orbit` gives the position vector of the Moon. It is constructed by `vector2XY` which takes the horizontal and vertical co-ordinates and yields the corresponding vector. Notice that the arguments are behaviours; the horizontal coordinate `(cos time)` is a behaviour that yields the cosine of the current time at all times; similarly for the vertical coordinate `sin time`.

The overall animation, `earthMoon`, is exactly as it reads; move an image of the Moon according to `orbit` and overlay it on an image of the Earth.

(Suitable definitions, such as imported bitmaps, are required for the `earth` and `moon` images.) The animation is viewed by entering

```
display earthMoon
```

and will run forever.

Let us extend `earthMoon` to obtain an animation of the Moon and Earth orbiting the Sun. The first step is to define a smaller version of the Earth and Moon animation;

```
smallEarthMoon = stretch 0.2 earthMoon
```

This can be put in orbit around the Sun, as follows:

```
sunEarthMoon = move orbit smallEarthMoon ‘over‘ sun
```

But in this animation the Earth and the Moon have the same orbital period, whereas we would like the Moon to orbit the Earth every month, or twelve times a year. We can do this by speeding up `smallEarthMoon` by a factor of twelve using the `faster` operator as follows:

```
sunEarthMoon = move orbit (faster 12 smallEarthMoon)
                   ‘over‘ sun
```

The above example shows that behaviours can be freely composed. This would not be the case for a naive implementation of the above animations in a procedural language in which one frame of the animation is produced with each iteration of a loop. More specifically, compositionality of the kind illustrated above is only possible if the following hold:

- Behaviours use continuous time. Discrete time representations will not compose straightforwardly when operations like `faster` are used.

- Behaviour expressions are pure and persistent. If they are not, side effects may interfere when behaviours are composed or reused.
- Behaviours are implicitly functions of time. If they depend on an explicit time variable then encapsulation is lost.
- Behaviours are structured values; for example, vectors.

3.2 Key concepts

The example program `sunEarthMoon` illustrates that animation components are compositional, and we identified some prerequisites for compositionality. In this section we will describe how the key concepts of Fran's approach lend themselves to compositional program construction, and other benefits they bring to developing animations.

Modelling. Writing programs that describe animations is often a difficult and time consuming task. Fran makes it easier by allowing authors to concentrate on content rather than on programming, or more precisely, on *what* the animation is rather than on *how* to display it on the screen. So Fran takes a declarative approach in which programs are models of animations. The implementation uses a presentation engine which computes how to display these models as animations. The presentation engine can be optimised by experts and then used by everyone, thus eliminating much duplication of effort.

The modelling approach is a compromise, however, because low-level control is lost and so animations that are not easy to describe within the modelling framework can be less efficient, and sometimes not possible. The choice of modelling framework is therefore crucial—it should be sufficiently expressive for most purposes but this must be balanced with the requirement to present models efficiently.

Continuous time. Animations are often represented as a sequence of frames which contain the image to display at discrete points in time. This is unnatural, however, because in the real world we usually regard motion as continuous; that is, objects move through a continuum of points in space in a continuous interval of time. Hence, it is more difficult to program animations in a language that uses discrete time representations, such as frames, than it is to model them mathematically. Fran adopts continuous time so that all time-varying values are conceptually functions of time. This has the following advantages:

- It is easier and more natural to describe time-varying values.
- Behaviours can always be composed, whereas with discrete time representations only behaviours that have the same points in time can be composed.
- Arbitrary time-transformations can be applied to any time-varying value, and this does not break compositionality.
- Motion can be described by rates of change using the differential calculus.
- It is possible, within certain limits, to run animations at approximately the same speed, regardless of the hardware, because frames can be computed at any point in time. The animation is unlikely to be as smooth on a slower machine, but after say five seconds the animation will be at (approximately) the same point as it would be on the faster machine.

Behaviours. In Fran behaviours are used to describe all time-varying aspects of animations. They are conceptually functions of time which map times to values of various types, depending on what is being described. For example, behaviours may yield colours, numbers, positions, shapes, images or sound. Behaviours are encapsulated—they cannot be evaluated at specific times (*sampled*) by the animator. Only the presentation engine can do this, so that it can compute frames of the animation [Ell99b]. This encapsulation is important because for certain behaviours to be efficient they must only be sampled monotonically—that is, at non-decreasing times. The presentation engine must guarantee to do this in order to obtain a reasonable level of efficiency. (In fact, there is a slight flaw in the design in this respect, because Fran provides time-transformations which allow behaviours to be speeded up, slowed down, delayed or in fact arbitrarily time warped. Consequently behaviours may not be monotonically sampled by the presentation engine. Pragmatically we need not regard this as a flaw; rather, we can expect the animator to be aware of this limitation and use time-transformations cautiously.)

Reactivity. Modelling in Fran is based on describing behaviours and how they react to events. This latter aspect is called *reactivity*. It allows us to describe interaction between the components of an animation; for example, collisions, timed events and user input.

A reactive behaviour is one that changes course when some criterion, called the event condition, is satisfied; for example, a ball's velocity behaviour that changes when the ball hits a wall. In this case, the event condition is that the surface of the ball is in contact with the wall. This condition is expressed using a boolean behaviour. Fran also has built in primitives for user input events such as mouse clicks.

Usually, what to do next depends on precisely which event occurred. Fran neatly captures this by packaging a new behaviour with the event condition, and then this new behaviour may be used when the event occurs. It is actually pairs comprising a condition and a new behaviour that Fran calls an event. Various event combinators are provided to manipulate these values. This results in a powerful and convenient notation for expressing reactivity.

Embedded language. Fran uses the embedded language approach; it is a library written in Haskell and animations are Haskell programs that import this library. This saves a lot of work designing and implementing a complete language, but also restricts the syntax and implementation to features available in the host language. As we have seen, Haskell syntax is superlative for Fran, but as an implementation language it is not very efficient. In summary, the convenience of the embedded language approach makes it ideal for prototyping and for proof of concept, but to create an efficient, industrial strength version would require an implementation in a more conventional language. Unfortunately, it is unlikely that the embedded language approach would work well for such implementations, because most other languages do not offer the syntactic convenience and expressive power of Haskell.

3.3 Time and Lifting

The behaviour `time` was used in Example 3.1 in the term `cos time`. We will explain how this term gives the behaviour that yields the cosine of the current time. Firstly, `time` is the behaviour that at all times yields the time, so it is the identity function on times. One way to specify the semantics of behaviours is to define a semantic function (at) which maps terms of type `Behaviour` θ (i.e., behaviours that yield values of type θ) to functions from times to values of type θ (we will denote the set of values of type θ by $[\![\theta]\!]$),

$$\mathrm{at}[\![\_]\!] : \texttt{Behaviour}\ \ \theta \rightarrow (\mathbb{T} \rightarrow [\![\theta]\!]).$$

Thus the semantic equation for `time` is,

$$\mathrm{at}[\![\texttt{time}]\!] \quad = \quad t \mapsto t.$$

That is, `time` represents the identity function on times.

Example 3.1 in Section 3.1 used the function `cos` that operates on behaviours. In Fran there is a uniform way of constructing such functions from non-behaviour functions called *lifting*. So, for example, the standard cosine function,

```
cos ::  RealVal -> RealVal
```

can be lifted to the behaviour level function, `cosB`, as follows:

```
cosB ::  Behaviour RealVal -> Behaviour RealVal
cosB = lift1 cos
```

The `cosB` function takes a real valued behaviour as its argument, and applies the cosine function to the value of this behaviour at all times. Thus, the semantic equation for `cosB` is

$$\mathrm{at}[\![\texttt{cosB}\ \ a]\!] \quad = \quad t \mapsto cos(\mathrm{at}[\![a]\!]t).$$

This idea applies to all functions, so in general we may lift a function

$$f \ \ \texttt{::} \ \ \phi \ \texttt{->} \ \theta$$

so that it operates on behaviours

$$\texttt{lift1}\ f \ \ \texttt{::} \ \ \texttt{Behaviour}\ \phi \ \texttt{->}\ \texttt{Behaviour}\ \theta$$

with the semantics that it applies the function f to the value of the behaviour argument a at all times,

$$\mathrm{at}[\![\texttt{lift1}\ f\ a]\!] = t \mapsto f(\mathrm{at}[\![a]\!]t).$$

Furthermore, lifting applies to constants and functions of any arity. The semantic equations for all the operators in Fran from Elliott and Hudak's semantics for Fran [EH97] are given in Section 3.7.

Lifting is also an important feature of CONTROL, and it is described in this context in Section 5.1.

In Example 3.1 we used the name `cos` for the behaviour level cosine function instead of `cosB`. This is possible in Fran because Haskell's type class mechanism is used to overload the names of many standard functions so that the behaviour level versions are used if the argument is a behaviour; that is, the overloading is resolved by the (inferred) argument type. Even numeric and other constants are overloaded this way.

3.4 Reactivity

Animations can be viewed as reactive systems where components of an animation, including the user, interact with each other. To accommodate reactivity, Fran provides an operator called `untilB` which can be used to construct a behaviour that changes course when a given event occurs. (Fran provides other operators for reactivity, but they are defined in terms of `untilB` which is the primitive operator.) Events have two parts; firstly, a condition which specifies when the event occurs, and secondly, a value associated with the event. The value part is usually the behaviour that will be used after the event has occurred. So, it is the combination of a condition and a value that Fran calls an event.

Events often require a user argument. For example, `lbp u` is the event that occurs when the left mouse button is pressed, and the user argument `u` is necessary to distinguish the next button press from previous ones. User arguments also supply start times for `integrals` and for `predicate` events (the time to start testing for the event occurrence) and are used by the implementation for passing sampling rates.

We can change the value part of an event using the `-=>` operator; for example, the condition part of `lbp u` is the event that occurs when the left mouse button is pressed, and the value part is the mouse release event. We can associate a different value with button presses as follows:

```
lbp u -=> red
```

This is the event that occurs when the left button is pressed, but yields the value `red` instead of the mouse release event.

Events are used in programs via the `untilB` operator. This takes a behaviour and an event:

`untilB : Behaviour` θ `-> Event (Behaviour` θ`) -> Behaviour` θ.

At first the given behaviour is used, but when the event condition first becomes true the behaviour switches to a new one obtained from the value part of the event. So, to obtain a behaviour that changes colour from blue to red when the left button is pressed, we pass the behaviour `blue` as the first argument to `untilB`, and then pass the event that occurs when the left mouse button is pressed and yields `red` as the second argument:

```
blue `untilB` (lbp u -=> red).
```

Events can be constructed from boolean behaviours using `predicate`; for example,

```
predicate (time >=* 5) u
```

is the event that occurs when the boolean behaviour `(time >=* 5)` first becomes true; that is, when the time is greater than or equal to 5. The operator `>=*` is the behaviour level greater than or equal to operator. Note the user argument `u` here. In this case it is used to give the time from when the condition should be tested, because in general predicate events should not be tested for all times since the animation began. Although it seems that user arguments are necessary in the Haskell-based implementation of Fran, they complicate reactive programs significantly. In CONTROL they have been eliminated giving a cleaner semantics and allowing a simpler programming style.

Events can be composed using operators such as `.|.` which chooses the earlier of two events, and yields the value associated with this event. For example, the event

```
lbp u .|. predicate (time >=* 5) u
```

occurs when either the left mouse button is pressed or the time reaches 5, whichever happens first. Unlike the logical OR operator, this behaviour level OR is asymmetric; if both events occur simultaneously then the new behaviour obtained is the first (left) argument. There are a number of other event operators, the details of which can be found in [EH97].

3.5 Integration for behaviours

Fran provides an operator called `integral` which, given a real valued behaviour a, yields the behaviour that gives the integral of a at all times. A user argument must also be supplied, which for integrals is used to determine the starting point of the interval to integrate over. For example, the integral of the behaviour `time` is given by

```
integral time u
```

for some user argument `u`. Assuming the user has a start time of zero, we can find the behaviour that is equivalent to this one by calculating the symbolic integral. The behaviour `time` corresponds to the identity function on times, $t \mapsto t$, and the symbolic integral of this function is $0.5 * t^2$. Therefore the following behaviour is equivalent to the one above:

```
0.5*time*time.
```

There are a number of subtleties concerning `integral`. Elliott and Hudak give the following definition in their semantics:

$$\text{at}[\![\texttt{integral}\ b\ t_0]\!]t \quad = \quad \int_{t_0}^{t} \text{at}[\![b]\!]x.dx$$

Note that in the original version of Fran user arguments were simply start times, so here t_0 is a time. In later versions the start time is extracted from the user argument. This semantics for `integral` fails to address a number of issues. Firstly, not all behaviours can be integrated so the expression on the right hand side is not always well-defined. Secondly, it does not define the semantics of recursive definitions using `integral`, which can have many solutions. (For example, the program

```
b = integral (b^(4/5)) 0
```

corresponds to the integral equation

$$y(t) = \int_{0}^{t} y(s)^{4/5}.ds.$$

This has many solutions, for example, $y(t) = 0$ or $y(t) = (1/5)^5 t^5$.) Thirdly, Elliott and Hudak's semantics does not explain how reactive behaviours can be integrated. We will return to these issues when we discuss integration in CONTROL.

3.6 Recursive behaviours

One of the most interesting features in Fran is that behaviours can be defined recursively by writing a recursive Haskell definition. This can be useful, and is sometimes necessary, when writing interactive animations. Upon reflection, this is as we would expect; if two objects interact with each other then their definitions must be in terms of each other. We will now give some examples.

The following program gives the position of a ball falling from height 1 to the ground at height 0. When it hits the ground it remains at rest:

```
h = 1 - integral 1 u `untilB` predicate (h <=* 0) u -=> 0
```

Here the condition when the ball hits the ground, `h <=* 0`, is in terms of the height, `h`, and so the definition is recursive.

We will now write a program which describes the path of body in orbit around a fixed body according to Newton's law of gravitational attraction. It could replace `orbit` in Example 3.1 to give a more realistic impression of the Moon's orbit around the Earth. Recall that Newton's inverse square law of gravitational attraction is

$$F = \frac{Gm_1m_2}{r^2}$$

G is the universal gravitational constant
m_1, m_2 are the masses of the bodies
r is the distance between the bodies

This gives the magnitude of the force on the Moon, and the direction of the force is always towards the Earth. This directed force is proportional to the acceleration of the Moon, so it can be integrated twice to give the position of the Moon. These formulas can be coded directly in Fran as follows:

```
orbit' u = s
  where
    s = s0 + integral v u
```

```
v = v0 + integral a u
a = (-k/(r ^ 2)) *^ unit_s

r = magnitude s
unit_s = (1 / mag_s) *^ s
```

(suitable values for the constant `k`, the initial position `s0`, and the initial velocity `v0`, are also required). Notice that the definitions of the position, `s`, the velocity, `v`, and the acceleration, `a`, are mutually recursive. We cannot avoid recursion if we want to use Newton's law of gravitation in this way because the position depends on the acceleration, but the gravitational force, and hence the acceleration, depends on the relative positions of the bodies.

As a final example, here is an animation of a ball following the mouse as if it were being dragged on a spring through a thick liquid,

```
followMouse u = move p ball
  where
    p = integral v u
    v = integral a u
    a = (mouseMotion u - p) - (0.5 *^ v)
```

Again, the position, velocity and acceleration are mutually recursive. This time it is because the force the spring exerts, and hence the acceleration of the ball, depends on the position of the ball relative to the mouse. It is not possible to solve the equations and write the position as an explicit formula, as it is, incidentally, for the previous example, because the mouse position is an input behaviour and is therefore not known beforehand. Therefore it is not possible to write this program in Fran without using recursion.

3.7 Semantics

Elliott and Hudak give a denotational semantics to the operators on behaviours and events, treating them as a pair of mutually recursive polymorphic datatypes [EH97]. This is not the same as giving a complete semantics to Fran, however, because Fran programs are written in Haskell and so they can be considerably more complicated than expressions using only the behaviour and event operators. In particular, behaviours may be defined by recursive definitions, and this is not accounted for by their semantics.

It may seem as if the semantics of Haskell is sufficient to determine the semantics of Fran because it is a Haskell library. However, such a semantics would be at the wrong level of abstraction—it would capture all the implementation details of behaviours but not their abstract nature. The presentation engine uses discrete sampling to compute values of behaviours at points in time, and computes integrals and event occurrences using numerical approximation techniques. Consequently a semantics based on the implementation would not give an exact semantics of behaviours.

So Elliott and Hudak's approach, giving a semantics to the operators on behaviours and events, combined with an understanding of Haskell's semantics, seems like a good first approximation to the semantics of Fran. However, the interaction of these abstract behaviours and events with Haskell, in particular with recursive definitions, cannot be explained by this approach. This is why our work takes a simpler language and provides a complete semantics for it.

If we ignore recursion, then Elliott and Hudak's semantics captures the abstract properties of behaviours and events. It does not capture the semantics of the implementation, however, because approximation techniques are used to compute integrals and event times. As we said in the Introduction, we should first try to give the values as if they were exact, and then consider the errors due to approximation. This second stage is necessary to be able to verify the correctness of implementations or to reason reliably about programs. For these purposes a more advanced theory that accounts for approximation is required. Our work does not deal with approximation either, but we take an idealised view of CONTROL which means that the language is defined to yield exact values of behaviours and event times. In other words, this assumption is made explicit rather than being ignored. Using this approach we are able to give a complete semantics for a functional language with behaviours, including a full treatment of recursion.

Elliott and Hudak's semantic function for behaviours assumes an abstract domain of polymorphic behaviours, $Behaviour_\alpha$. These abstract behaviours are interpreted as functions from times to values by the semantic function 'at' which we used previously:

$$\mathrm{at} : Behaviour_\alpha \to Time \to \alpha.$$

The intention is that these abstract behaviours correspond to the behaviours that can be constructed in the Haskell based implementation. As we said above, this correspondence is not exact because the implementation computes approximations to the abstract behaviours described by this semantics.

Events belong to the abstract domain $Event_\alpha$ and are interpreted as $Time \times \alpha$ pairs by the semantic function occ:

$$\mathrm{occ} : Event_\alpha \to Time \times \alpha.$$

Recall that an event occurs at some time and yields a value which, for reactive behaviours, is the behaviour that will be used after the event has occurred. There are two problems that must be addressed here:

- An event might never occur and so it does not have an event time.

- It is not possible to see into the future to find when an event occurs, so a reactive behaviour cannot be specified in terms of the event time. More precisely, at time t a reactive behaviour only needs to know whether the event has occurred, and does not require the actual time of the occurrence. This is vital for external events, such as mouse clicks, which cannot be predicted ahead of time.

The first problem is solved by Elliott and Hudak by adding an infinite time, ∞, to the set of real numbers to represent the time of an event that never occurs. The second problem is solved by defining an ordering on times such that a time t is less than an event time t_e if either t_e is known and is greater than t, or else it is known to be at least as great as t.

The typing constraints for operators on abstract behaviours are given in Figure 3.1. These correspond precisely to the types of the operators in Haskell. The semantic equations for these operators are shown in Figure 3.2. They are straightforward interpretations of the operators we have already seen in the preceding sections, although there are some subtleties with `untilB` which we will discuss later on when we compare its semantics to `until-then` in CONTROL.

For us the most important operation on events is *predicate* which allows an event to be specified by a boolean behaviour. This is the only kind of event that is available in CONTROL because it does not provide for external inputs such as mouse clicks. The semantics of *predicate* is given by

$$predicate : Behaviour_{Bool} \rightarrow Time \rightarrow Event_{()}$$

$$\mathrm{occ}[\![predicate\ b\ t_0]\!] = (\inf\{t > t_0 \mid \mathrm{at}[\![b]\!] t\}, ())$$

So the time of a predicate event in Fran is the infimum of the set of times greater than t_0 when b is true. This is different from our treatment of event

$$time : Behaviour_{Time}$$

$$lift_n : (\alpha_1 \rightarrow \ldots \alpha_n \rightarrow \beta) \rightarrow$$

$$\quad Behaviour_{\alpha_1} \rightarrow \ldots \rightarrow Behaviour_{\alpha_n} \rightarrow Behaviour_{\beta}$$

$$timeTransform : Behaviour_{\alpha} \rightarrow Behaviour_{Time} \rightarrow Behaviour_{\alpha}$$

$$integral : VectorSpace\ \alpha \Rightarrow Behaviour_{\alpha} \rightarrow Time \rightarrow Behaviour_{\alpha}$$

$$untilB : Behaviour_{\alpha} \rightarrow Event_{Behaviour_{\alpha}} \rightarrow Behaviour_{\alpha}$$

Figure 3.1: Types of abstract behaviours in Fran

$$\begin{array}{lcl}
\mathrm{at}[\![time]\!]t & = & t \\
\mathrm{at}[\![lift_n\ f\ b_1 \ldots b_n]\!]t & = & f\ (\mathrm{at}[\![b_1]\!]t) \ldots (\mathrm{at}[\![b_n]\!]t) \\
\mathrm{at}[\![timeTransform\ b\ tb]\!] & = & \mathrm{at}[\![b]\!] \circ \mathrm{at}[\![tb]\!] \\
\mathrm{at}[\![integral\ b\ t_0]\!]t & = & \int_{t_0}^{t} \mathrm{at}[\![b]\!]x.dx \\
\mathrm{at}[\![b\ untilB\ e]\!]t & = & \text{if } t \leq t_e \text{ then } \mathrm{at}[\![b]\!]t \text{ else } \mathrm{at}[\![b']\!]t \\
\quad \text{where } (t_e, b') = \mathrm{occ}[\![e]\!] & &
\end{array}$$

Figure 3.2: Semantics of abstract behaviours in Fran

times of predicate events, and the differences will be explored in detail later on. The value associated with a predicate event is the unit value, ().

There are other operators on events in Fran, most importantly, for specifying external events and for handling the values associated with events. These are useful when programming with Fran, but are not relevant to CONTROL because it does not have external events (all events are like Fran's *predicate* events) and there are no values associated with events. The semantics of these operators is given in [EH97].

3.8 Summary of the literature

We will now give a brief summary of the literature on Fran. Many of the ideas behind Fran were developed in previous work, particularly TBAG [SEYAE94, ESYAE94], MediaFlow [ESAE95] and Active VRML [Ell96] (Fran is a concrete realisation of the ideas in [Ell96]). The seminal paper by Elliott and Hudak is [EH97]. This gives the key ideas, a semantics for the operators, and some details of the implementation. The language is described emphasising the embedded language approach in [Ell97, Ell99a].

There are many tutorials, applications and examples. Elliott has written a tutorial [Ell98a], and two extended applications which describe two-handed image navigation [Ell98e] and a fifteen puzzle [Ell98c]. The method of programming with events in Fran is described in [Ell98b]. Thompson uses Fran to program a lift simulation [Tho98]. In Chapter 9 we give a lift simulation in CONTROL which is much simpler, and thus illustrates the improved semantics of CONTROL compared to Fran. Daniels's tutorial paper constructs an animation of crew rowing [Dan97a].

Aspects of the Haskell based implementation are discussed in [Ell98d, Ell99b]. Fran has also been extended for robots; see [Lin98, PHE99]. Finally, some very preliminary work on a semantics for Fran is presented in [Lin97].

Chapter summary

Fran is the primary inspiration for CONTROL. It is intended for programming animations, but with suitable extensions it could also be used for implementing many other kinds of reactive systems. There are four key concepts that distinguish it from many other reactive languages: modelling, continuous time, behaviours and reactivity. The implementation uses the embedded language approach with Haskell as the host language.

Fran provides behaviours for representing time-varying quantities and events for expressing reactivity. In addition, recursive behaviours can be written using standard Haskell definitions. This offers an elegant and powerful programming technique. However, the semantics of Fran, and in particular of recursively defined behaviours, is not well developed. Elliott and Hudak have given a semantics to the operators on behaviours and events, but this does not account for recursively defined behaviours, nor for the approximation methods used in the implementation.

Chapter 4

A language for behaviours

In this chapter we introduce a new language called CONTROL. The development of this language and of its formal semantics are the principal subjects of the remainder of this dissertation.

As we said in the introduction, CONTROL is a functional language supplemented with operators for describing behaviours. These operators are inspired by similar operators in Fran. Where they differ from Fran is firstly due to some simplifications we have made and secondly due to improvements we have discovered while investigating the semantics.

We begin by introducing the syntax of CONTROL followed by the domains that values of each type belong to. This provides a starting point for the more detailed discussions of the semantics that follow.

4.1 Syntax

The functional core of CONTROL is a subset of PCF [Sco93] that includes simply typed λ-terms, a recursion operator and built in numbers. Like PCF, it uses normal order evaluation. The syntax of this functional core is as follows:

$$K \in \text{Constants}, \qquad x \in \text{Variables}$$

$$\text{Types} \quad \theta \quad ::= \quad \texttt{Real} \mid \texttt{Bool} \mid \theta \;\texttt{->}\; \theta$$

$$\text{Terms} \quad E \quad ::= \quad K \mid x \mid \lambda x\!:\!\theta\,.\,E \mid E\,E \mid \mu x\!:\!\theta\,.\,E$$

Numbers in CONTROL are real numbers rather than integers. The set of constants includes arithmetical operators (`+`, `-`, `*` etc.) and logical operators (`not`, `or`, `and` etc.). Notice that there are no explicit operations for pairs, and no `if-then-else` construct; these features are accounted for by constants.

Both λ and μ bind a variable within a term; λ is for λ-abstractions and μ is for recursive definitions. These are explained in detail in Chapter 7. The type of the variable must be supplied for both these binding constructs. So, for example,

$$\lambda x : \theta.E$$

means that the variable x has type θ and is bound by λ within E (and similarly for μ).

The type system for this fragment is very straightforward—it is as for the simply typed λ-calculus with the standard rule for μ. The typing rules are given in Chapter 8 as part of the complete formal description of the language. Note that only well typed terms are meaningful. Also, there are no type annotations other than those for bound variables. We will sometimes state the type of a term—for instance, $E : \theta$ asserts that the term E has type θ—but this is meta-notation and not valid syntax.

The remaining operators in CONTROL are for constructing behaviours, which are values of type `Beh` θ for some type θ. The behaviour operators extend the grammar as follows:

$$
\begin{array}{llll}
\text{Types} & \theta & ::= & \mathrm{Beh}\ \theta \\
\text{Terms} & E & ::= & \texttt{lift0}\ E \mid E\ \texttt{\$*}\ E \mid \texttt{integral}\ E \mid \\
 & & & E\ \texttt{until}\ E\ \texttt{then}\ E \mid \nu x{:}\theta.E
\end{array}
$$

Here ν binds a variable x within a term E, and, as for λ and μ, a type for x must be supplied. We will discuss the purpose and semantics of these behaviour operators later on. In this chapter we consider the domains values belong to, so it suffices at this stage to know that behaviours represent functions from times to values.

4.2 Domains

Our aim is to define the mathematical meaning of all CONTROL programs; that is, to define for every valid term a value that denotes its meaning. This is called a denotational semantics and is usually defined compositionally, which means that the value of a term is constructed from the values of its immediate subterms. This is a very economical method because all that is required is a general formula for each syntactic construct (i.e., for each production of the abstract grammar) and then the meaning of every term in the language can be obtained.

Firstly we must state what domains these values belong to. This is a vital step because it is sometimes uncertain whether the domains we assume by our semantic equations actually exist. For example, it was not known for around three decades whether a domain existed for the untyped λ-calculus. (This is difficult because functions and arguments belong to the same domain, and in set theory function spaces are always strictly larger than the domains they map between. CONTROL uses the simple type system and consequently avoids this problem.) Furthermore, describing the domains reveals a lot about a language. As Christopher Strachey advised: "I think it would be well worth the effort of any language designer to start with a consideration of the domain structure" [Str73].

For a typed language like CONTROL we require a domain corresponding to each type. Types are either ground types—`Real` or `Bool`—or else composite types—functions or behaviours. We will begin with domains for the ground types.

Terms of type `Bool` represent truth values—either true or false. These two values form the set of boolean values,

$$\mathbb{B} \quad = \quad \{\mathrm{true}, \mathrm{false}\}.$$

However, in most programming languages, including CONTROL, we can write terms that are type correct but do not terminate—they get stuck in an infinite loop. We need a value to denote such terms; for the domain corresponding to `Bool` we will use the symbol $\perp_{\mathbb{B}}$. Other domains also use the symbol $\perp$ with different subscripts to denote non-terminating terms, and such a value is called bottom. We indicate domains by enclosing types in semantic braces $[\![\_]\!]$, so for the type `Bool` we have

$$[\![\texttt{Bool}]\!] \quad = \quad \mathbb{B} \cup \{\perp_{\mathbb{B}}\}.$$

For convenience we will denote any domain formed by adding a bottom element $\perp_A$ to a set A by $A_\perp$; hence,

$$\mathbb{B}_\perp \quad = \quad \mathbb{B} \cup \{\perp_{\mathbb{B}}\}.$$

Similarly, terms of type `Real` represent either real numbers or non-termination:

$$[\![\texttt{Real}]\!] \quad = \quad \mathbb{R} \cup \{\perp_{\mathbb{R}}\} = \mathbb{R}_\perp.$$

Many languages use a floating point representation for real numbers, and then operations on them yield approximate results. In contrast, CONTROL is an idealised language where all operations on real numbers are exact and there is no overflow.

We now turn our attention to functions. They require a more complex domain structure than ground types, but the theory we make use of is well established. Our exposition here explains why we need to use this theory, gives all the necessary definitions and provides some intuition for the motivation of the theory, but a detailed analysis is beyond our scope. This can be found in any standard text covering denotational semantics [Rey98, Sto77], or the original papers by Scott and Strachey [Sco70b, Sco70a, Sco76, SS71].

A function in a programming language is really an algorithm that will be performed each time the function is applied to an argument. For each possible argument the algorithm will either produce a value or else loop indefinitely; either way the result will be an element in the domain corresponding to the result type of the function. Thus, it appears that a suitable domain for function types is actual functions between domains, that is,

$$[\![\theta \texttt{ -> } \theta']\!] \quad = \quad [\![\theta]\!] \rightarrow [\![\theta']\!].$$

This equation defines the domain for functions of type $\theta\texttt{->}\theta'$ to be all set theoretic functions from $[\![\theta]\!]$ to $[\![\theta']\!]$. Many of these functions are uncomputable, however, and so the functions that we can represent in CONTROL constitute a small subset of this domain. This is not a problem because in general we require a domain $[\![\theta]\!]$ to contain a value for every valid term of type θ, but it does not have to be the smallest such domain. However, we can benefit from removing some unwanted values from function domains because doing so makes it easier to define the meaning of recursive definitions. In the simply typed λ-calculus, this is the only reason for adopting a more complicated model than the basic set theoretic one. We will now describe how we restrict function domains, and in Chapter 7 we will show how these restricted domains allow us to assign meanings to recursive definitions.

The restriction of function domains is based on a notion of how defined functions are. A function g is at least as defined as a function f if it is at least as defined for all arguments. This is called the pointwise ordering on function spaces. For example, consider functions in the domain $\mathbb{B}_\perp \to \mathbb{B}_\perp$. We will write t, f and $\perp$ as shorthands for $true$, $false$ and $\perp_\mathbb{B}$, and write functions,

$$[t : x \mid f : y \mid \perp : z],$$

as ordered tuples,

$$(x, y, z).$$

The function $(\perp, \perp, \perp)$ is the least defined of all the 3^3 functions in $\mathbb{B}_\perp \to \mathbb{B}_\perp$. Both $(t, \perp, \perp)$ and $(\perp, f, \perp)$ are more defined than this least element, and in turn $(t, f, \perp)$ is more defined than both these functions (it is the identity function on $\mathbb{B}_\perp$). Note that $(t, \perp, \perp)$ is neither more nor less defined than $(\perp, f, \perp)$; they are incomparable. Therefore this ordering on functions is a partial order. Partial orders can be drawn using Hasse diagrams, so for the above example we have,

$$\begin{tikzcd}
& (t, f, \perp) \arrow[dl] \arrow[dr] & \\
(t, \perp, \perp) \arrow[dr] & & (\perp, f, \perp) \arrow[dl] \\
& (\perp, \perp, \perp) &
\end{tikzcd}$$

This ordering on $\mathbb{B}_\perp \to \mathbb{B}_\perp$ requires the following simple ordering on $\mathbb{B}_\perp$: $\perp_\mathbb{B}$ is less than $true$ and $false$, but $true$ and $false$ are incomparable, that is,

$$x \sqsubseteq_{\mathbb{B}_\perp} y \iff x = \perp_\mathbb{B} \vee x = y.$$

The Hasse diagram for this is

$$\begin{tikzcd} true \arrow[dr] & & false \arrow[dl] \\ & \perp_\mathbb{B} & \end{tikzcd}$$

The partial order for the domain $\mathbb{R}_\perp$ is defined similarly.

Given these orderings on ground types we can define an ordering for any domain $D \to D'$ assuming that we have an ordering on D'. This is the pointwise order mentioned above:

$$f \sqsubseteq_{D \to D'} g \iff \forall x \in D : fx \sqsubseteq_{D'} gx.$$

We obtain an ordering on all function types by induction. Because this approach is so common in programming language semantics it is usual to use the term domain to refer to both the set of values for a given type and the partial order on that set. This is sensible because we are about to restrict the domains for function types using the partial order.

For functions over infinite sets, such as functions on integers, the partial order $\sqsubseteq_{\mathbb{Z}_\perp \to \mathbb{Z}_\perp}$ gives infinite sequences of increasingly defined functions, called ω-chains. For example, consider the following chain of increasingly defined functions that approximate the identity function on integers:

$$\begin{array}{c} id_{\mathbb{Z}_\perp} \\ | \\ \vdots \\ | \\ [n \mapsto \perp|2:2|1:1|0:0] \\ | \\ [n \mapsto \perp|1:1|0:0] \\ | \\ [n \mapsto \perp|0:0] \\ | \\ [n \mapsto \perp] \end{array}$$

The importance of ω-chains is that we require all our domains to be ω-chain complete (or ω-complete), which means that all ω-chains must have a least upper bound. An upper bound of a set is an element in the domain that is greater than all the elements in the set, and a least upper bound is the least such element. If it exists, we denote the least upper bound of a set X by $\bigsqcup X$. Thus, for a domain D,

$$D \text{ is } \omega\text{-complete} \iff \forall\ \omega\text{-chains } (x_i) \in D : \bigsqcup_D (x_i) \in D \text{ exists.}$$

Both $\mathbb{B}_\perp$ and $\mathbb{R}_\perp$ are ω-complete because the only chains are trivial ones such as $\{\perp_\mathbb{B}, true, true, \dots\}$. Function spaces are not ω-complete in general, so they need restricting.

The condition we use to restrict function spaces is ω-continuity (sometimes called Scott-continuity). This is sufficient to ensure that the domains for function spaces are ω-complete. In fact it is a stronger condition than is necessary, but this does not matter because all computable functions are ω-continuous, and so placing this condition does not eliminate any useful values from function domains. ω-continuity requires that functions preserve least upper bounds; that is,

$$f \text{ is } \omega\text{-continuous} \iff \forall\ \omega\text{-chains } (x_i) : \bigsqcup_{D'} f(x_i) = f(\bigsqcup_D (x_i))$$

To re-iterate, assuming that D and D' are ω-complete, we can show that the subset of all functions $D \to D'$ comprising ω-continuous functions is ω-complete. Therefore if all function spaces are limited to ω-continuous functions then all domains are ω-complete by induction. Such domains are called complete partial orders (CPOs), or pointed CPOs when they have a least element. Our domains are pointed CPOs because they are all ω-complete, and the least elements of the domains $\mathbb{B}_\perp$, $\mathbb{R}_\perp$ and $[\![\theta\texttt{->}\theta']\!]$ are $\perp_\mathbb{B}$, $\perp_\mathbb{R}$ and $x \mapsto \perp_{[\![\theta']\!]}$ respectively. Finally, we will use the same notation for ω-continuous functions as for functions on sets; that is,

$$[\![\theta \texttt{ -> } \theta]\!] \quad = \quad [\![\theta]\!] \to [\![\theta']\!]$$

Here the arrow on the right hand side denotes ω-continuous functions because $[\![\theta]\!]$ and $[\![\theta']\!]$ are domains. In fact, this notation is consistent with the usual notation using $\to$ for arbitrary functions between sets so long as we assume a discrete order (i.e., $x \sqsubseteq y \iff x = y$) on sets because with a discrete order there are no non-trivial infinite chains and hence all functions are ω-continuous.

The properties that all domains are pointed CPOs and that all functions are ω-continuous are sufficient to apply the least fixed point theorem in order to obtain values for all recursive definitions. We will take this approach in Chapter 7 when we discuss recursive functions.

4.3 Domains for behaviours

In this section we discuss the domains for behaviour types. We can write functions that accept and yield behaviours, so to be consistent with our interpretation of function domains we must use pointed CPOs for behaviour domains.

Behaviours represent functions from times to values, which suggests the following domain equation:

$$[\![\texttt{Beh }\theta]\!] \;=\; \mathbb{T} \to [\![\theta]\!].$$

Here $\to$ means all functions from the set of times, $\mathbb{T} = \{t \in \mathbb{R} \mid t \geq 0\}$, to the underlying set of the domain $[\![\theta]\!]$. The least defined member of this domain is the one that maps all times to $\perp$. We define the information order on behaviours to be a flat order, as for `Bool` and `Real`,

$$a \sqsubseteq_{\texttt{Beh }\theta} b \quad\Longleftrightarrow\quad a = t \mapsto \perp_{[\![\theta]\!]} \vee a = b$$

At first sight this ordering appears too simplistic; we are used to pointwise orderings on domains for function spaces. But behaviours are special representations of functions—abstract values like real numbers—and all that is required is that they satisfy certain operations. Although behaviours represent functions from times to values it is not possible to evaluate behaviours at particular times in CONTROL—this would break their abstract representation. Furthermore, we have a different interpretation of recursive behaviours to recursive functions, so we do not need to apply the least fixed point theorem for recursive behaviour definitions and therefore they do not have to be ω-continuous functions. For these reasons, this simple domain is sufficient for our semantics.

Another concern is that if some behaviours are not ω-continuous functions then they are not computable, because all computable functions are ω-continuous. This is irrelevant because we are taking an idealised view of behaviours, assuming that we can compute various operations over them.

Finally, notice that the domain with the given order $\sqsubseteq_{\texttt{Beh }\theta}$ is ω-complete, which is essential if we are going to write recursive functions over behaviours.

4.4 Semantic functions

We have established the domains that values denoting the meaning of CONTROL terms belong to. The next step is to define the meaning of every term by providing mappings from terms to values. These mappings are called semantic functions and are usually defined compositionally; in other words, the value of a term is constructed from the value of its immediate syntactic subterms. This way, provided that we have a formula for each syntactic construct, we can obtain the meaning of any term in the grammar.

Semantic functions must yield values in the appropriate domain, so a term of type θ must be mapped to a value in the domain $[\![\theta]\!]$. Also, formulas given by semantic functions must be type correct. This is straightforward in CONTROL because the simple type system constructs the type of a term from the types of its subterms, and so long as each semantic equation is type correct a well typed value will result for any well typed term. Furthermore, there is at most one valid typing for any term in CONTROL, so we may omit type information from our semantic equations without ambiguity.

We will write $[\![\_]\!]$ for all semantic functions. Semantic braces are useful because they separate the object-language from the meta-language, and we prefer to avoid clutter and not name our semantic functions. We will define a family of functions, one for each type, and overload $[\![\_]\!]$ by using it for all these functions.

Constants of type `Bool` and `Real` correspond to values in the obvious way:

$$[\![\texttt{true}]\!] \;=\; true,$$

(and similarly for false),

$$[\![\texttt{0}]\!] \;=\; 0,$$

(and similarly for all real numbers). The boolean constants could be used in a concrete syntax for CONTROL, but the syntax of real numbers is more difficult to define. However, we are using abstract syntax and so the actual representation is unimportant. Therefore we can use the usual decimal notation for real number terms, so numbers in the object-language and in the meta-language have the same representation.

In addition to constants there are many built in functions on `Real` and `Bool` types. These functions represent the usual mathematical functions extended to yield $\perp$ when applied to $\perp$, or when they are undefined. For example, the built in function `sin` satisfies the following definition:

$$[\![\texttt{sin}\ E]\!] \;=\; \left\{ \begin{array}{l} \perp_{\mathbb{R}} \\ sin[\![E]\!] \end{array} \right. \left| \begin{array}{l} [\![E]\!] = \perp_{\mathbb{R}} \\ \end{array} \right.$$

(the function sin on the right hand side is the usual mathematical one whose domain is the real numbers). Some functions are not defined for every value in their domain, and they yield bottom at these values. For example, division is undefined when the second argument is zero, thus,

$$[\![A\ \texttt{/}\ B]\!] \;=\; \left\{ \begin{array}{l} \perp_{\mathbb{R}} \\ \perp_{\mathbb{R}} \\ [\![A]\!]/[\![B]\!] \end{array} \right. \left| \begin{array}{l} [\![A]\!] = \perp_{\mathbb{R}} \vee [\![B]\!] = \perp_{\mathbb{R}} \\ [\![B]\!] = 0 \\ \end{array} \right.$$

All built in functions extend the usual ones in this way. To avoid cluttering the notation, we will use the usual names for these extended functions. Thus, when we write sin or $/$ we are referring to the function defined by the right hand side of the above equations. Using this convention, here are the equations for logical-and and addition:

$$\begin{aligned} [\![\texttt{and}]\!] &= \wedge \in \mathbb{B}_\perp \to \mathbb{B}_\perp \to \mathbb{B}_\perp \\ [\![\texttt{+}]\!] &= + \in \mathbb{R}_\perp \to \mathbb{R}_\perp \to \mathbb{R}_\perp \end{aligned}$$

Functions such as these which always yield $\perp$ when any argument is $\perp$ are called *strict* functions.

There is one exception, the function `if-then-else`, which is not strict in all its arguments. It is defined as follows:

$$[\![\texttt{if } C \texttt{ then } D \texttt{ else } E]\!] \quad = \quad \left\{ \begin{array}{l|l} \perp & [\![C]\!] = \perp_{\mathbb{B}} \\ [\![D]\!] & [\![C]\!] = true \\ [\![E]\!] & [\![C]\!] = false \end{array} \right.$$

This function is strict in C, but not in D or E. This is the case in virtually all programming languages because if C is true then it does not matter whether E terminates, and similarly if C is false then it does not matter whether D terminates.

The semantic functions for other constructs in the language are far more complicated than for constants. Chapters 5 to 7 introduce these functions for different parts of the language. We will finish this section with one special behaviour construct, `time`, which is the behaviour that yields, for every time, the current time:

$$[\![\texttt{time}]\!] \quad = \quad t \mapsto t \ \in \mathbb{T} \to \mathbb{R}_{\perp}.$$

Chapter summary

The syntax of CONTROL can separated into the non-behaviour fragment, which is very close to PCF, and the behaviour operators, which extend this fragment. CONTROL uses a minor extension to the simple type system.

A denotational semantics requires a domain corresponding to each type. For ground types and function types these are standard CPOs. The domain for behaviours has a flat ordering because behaviours are an abstract type and therefore do not need a more complex structure.

Chapter 5

Behaviour expressions

Behaviour expressions represent functions from times to values. We have already seen one example, the behaviour *time*, which is the identity function on times. CONTROL has a uniform way of lifting values to behaviours, inspired by Fran, and this makes it possible to apply all existing functions to behaviours. Other functions of time are constructed using the primitive operators for reactivity and integration. This chapter introduces these operators and develops a semantics for them. Subsequent chapters will then explore other aspects of the language. Our exposition follows an incremental development of the language and its semantics in unison, which helps to motivate the eventual definitions. The final description of the syntax and semantics of the language appears in Chapter 8: Complete formal semantics.

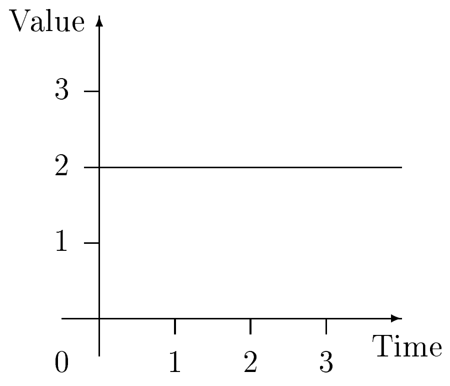
5.1 Lifting

In our context, lifting means turning values into behaviours. For the case of constants, $c : \theta$, this involves making the behaviour that yields c at all times—that is, a representation of the constant function $t \mapsto [\![c]\!]$. For the case of functions, $f : \phi$->θ, lifting involves making the function on behaviours that applies f to behaviours at all times. In other words, lifting a function is similar to mapping a function over a list of values, except that the 'list' is continuous instead of discrete because there is a value for every time.

The operator that lifts constants is `lift0`. It can be used to lift real numbers, for example,

$$[\![\texttt{lift0 2}]\!] \quad = \quad t \mapsto 2 \qquad \in \mathbb{T} \to \mathbb{R}_{\perp}.$$

Real-valued behaviours can be illustrated using graphs where points on the graph represent the value of the behaviour at each time (with time on the horizontal axis). The above example gives the graph which is a horizontal line through 2;



Note that the set-theoretic function $t \mapsto 2$ is an element of the domain $\mathbb{T} \to \mathbb{R}_{\perp}$ as required.

Values of any type can be lifted with `lift0`—it is a polymorphic operator. The semantics of `lift0` is:

$$\begin{aligned} \texttt{lift0} \quad &: \quad \forall\theta.\theta \to \texttt{Beh } \theta \\ [\![\texttt{lift0 } x]\!] \quad &= \quad t \mapsto [\![x]\!]. \end{aligned} \tag{5.1}$$

CONTROL does not have polymorphic data types, however. In other words, the type system does not permit values that have polymorphic types, which would require a more sophisticated type system. Polymorphic operators that are built in, such as `lift0`, are not difficult to incorporate because they are dealt with explicitly by the type rule for the operator.

Later on when we give the full typing rules we will see that the type system restricts arguments to `lift0` to non-behaviour types. Lifting behaviours is not very useful. Consider `lift0` b for some behaviour b. The meaning of a behaviour is determined by the values it yields for all times, but in this case the values are behaviours and so they also yield values at all times. However, the value could have been defined as the value of the overall behaviour directly rather than indirectly via another behaviour. In short, behaviours provide temporal abstraction, and behaviours of behaviours do not add any useful expressiveness.

Returning to Equation 5.1 we have that x must be a non-behaviour term. This means that the semantic function $[\![\_]\!]$ on the right hand side of Equation 5.1 maps non-behaviour terms to values. At this stage the only non-behaviour terms we can construct are constants, so this semantic function is the trivial mapping given in the previous chapter (i.e., the one that maps representations of constants and built-in functions to their mathematical counterparts). Later on we will introduce facilities for writing new functions in CONTROL, and then we will need to extend this semantic function accordingly.

Given a function, we may want to apply it to a behaviour by mapping it over the behaviour at all times. For example, say we want to construct the behaviour whose graph is a sine wave; one way to do this is to apply the function `sin` to the behaviour `time` at all times. This is exactly what the operator `lift1` enables us to do:

$$\begin{aligned}
[\![\texttt{lift1 sin time}]\!] &= t \mapsto sin([\![\texttt{time}]\!]t) \\
&= t \mapsto sin((t \mapsto t)t) \\
&= t \mapsto sin(t).
\end{aligned}$$

In general, `lift1` can be used to map a function $f_1 : \phi\texttt{->}\theta$ over a behaviour $x : \texttt{Beh}\ \phi$ to yield a new behaviour $\texttt{lift1}\ f_1\ x : \texttt{Beh}\ \theta$. This new behaviour gives, at any time t, $[\![f_1]\!]$ applied to the value of the behaviour x at time t; that is,

$$\begin{aligned}
\texttt{lift1} &: (\phi\ \texttt{->}\ \theta)\ \texttt{->}\ \texttt{Beh}\ \phi\ \texttt{->}\ \texttt{Beh}\ \theta \\
[\![\texttt{lift1}\ f_1\ x]\!] &= t \mapsto [\![f_1]\!]([\![x]\!]t) \qquad (5.2) \\
&= [\![f_1]\!] \circ [\![x]\!].
\end{aligned}$$

We should check that this equation is type correct. Recall that a value of type $\texttt{Beh}\ \phi$ belongs to the domain $\mathbb{T} \to [\![\phi]\!]$, and so

$$[\![x]\!]t \in [\![\phi]\!].$$

This is a valid argument for $[\![f_1]\!]$ because

$$[\![f_1]\!] \in [\![\phi]\!] \to [\![\theta]\!]$$

and the thus the values on both sides of Equation 5.2 belong to the domain $\mathbb{T} \to [\![\theta]\!]$.

Functions of any arity can be lifted in a similar way to unary functions. For functions with two arguments, $f_2 : \phi_1\texttt{->}\phi_2\texttt{->}\theta$, we require a primitive operator `lift2`, with the following semantics:

$$\texttt{lift2} : (\phi_1\ \texttt{->}\ \phi_2\ \texttt{->}\ \theta)\ \texttt{->}\ \texttt{Beh}\ \phi_1\ \texttt{->}\ \texttt{Beh}\ \phi_2\ \texttt{->}\ \texttt{Beh}\ \theta$$

$$[\![\texttt{lift2}\ f_2\ x_1\ x_2]\!] = t \mapsto [\![f_2]\!]([\![x_1]\!]t)([\![x_2]\!]t).$$

So, for example, we obtain pointwise addition of real-valued behaviours by applying `lift2` to `+` : `Real->Real->Real`, as follows:

$$[\![\texttt{lift2 + } x \;\; y]\!] \;\; = \;\; t \mapsto ([\![x]\!]t) + ([\![y]\!]t).$$

The (+) on the right hand side is the one described towards the end of Section 4.2.

In general, a function of arity n,

$$f_n : \phi_1 \texttt{ -> } \ldots \texttt{ -> } \phi_n \texttt{ -> } \theta,$$

can be lifted to obtain the behaviour level version,

$$\texttt{lift}\langle n\rangle \;\; f_n : \texttt{Beh } \phi_1 \texttt{ -> } \ldots \texttt{ -> Beh } \phi_n \texttt{ -> Beh } \theta,$$

using the lifting operator $\texttt{lift}\langle n\rangle$, with the following semantics:

$$[\![\texttt{lift}\langle n\rangle \;\; f_n \;\; x_1 \;\; \ldots \;\; x_n]\!] \;\; = \;\; t \mapsto [\![f_n]\!]([\![x_1]\!]t)\ldots([\![x_n]\!]t).$$

There is a simplification which allows all these lifting operators to be expressed in terms of `lift0` and a "lifted application" operator, `$*`. To show how this works, we will express `lift1` in terms of `lift0` and `$*`.

Firstly, recall the definition of `lift1`,

$$[\![\texttt{lift1} \;\; f_1 \;\; x]\!] \;\; = \;\; t \mapsto [\![f_1]\!]([\![x]\!]t) \tag{5.3}$$

Now, because `lift0` is polymorphic, we can apply it to $f_1 : \phi\texttt{->}\theta$ to obtain the constant valued behaviour that yields $[\![f_1]\!]$ at all times,

$$[\![\texttt{lift0} \;\; f_1]\!] = t \mapsto [\![f_1]\!].$$

From this it follows that

$$[\![\texttt{lift0} \;\; f_1]\!]t \;\; = \;\; [\![f_1]\!]. \tag{5.4}$$

Next we define, as a primitive, a lifted function application operator, `$*`

$$[\![fb\ \texttt{\$*}\ x]\!] \;=\; t \mapsto ([\![fb]\!]t)([\![x]\!]t) \tag{5.5}$$

This takes on the left, a behaviour that yields functions, and on the right, a behaviour that yields arguments (of the appropriate type) and applies the functions to the arguments at each time. To define `lift1` in terms of `lift0` and `$*` we reason as follows:

$$\begin{aligned}
& [\![\texttt{lift1}\ f_1\ x]\!] \\
= \;& \langle \text{by } (5.3) \rangle \\
& t \mapsto [\![f_1]\!]([\![x]\!]t) \\
= \;& \langle \text{by } (5.4) \rangle \\
& t \mapsto ([\![\texttt{lift0}\ f_1]\!]t)([\![x]\!]t) \\
= \;& \langle \text{by } (5.5), \text{ with } fb = \texttt{lift0}\ f_1 \rangle \\
& [\![\texttt{lift0}\ f_1\ \texttt{\$*}\ x]\!].
\end{aligned}$$

So `lift1` can be defined in terms of `lift0` and `$*`, and in general,

$$\texttt{lift}\langle n\rangle\ f_n\ x_1\ \ldots\ x_n \;=\; \texttt{lift}\langle n-1\rangle\ f_n\ x_1\ \ldots\ x_{n-1}\ \texttt{\$*}\ x_n$$

so any lifting operator can be defined in terms of `lift0` and `$*`. This is useful in practice because it reduces the number of primitives in our language which in turn reduces the number of semantic equations.

5.2 Reactivity

In CONTROL behaviours may react or change course in response to *events*; following Fran, we call this *reactivity*. In our context, event conditions are defined using boolean behaviours, and there is no facility for external events.

A practical language based on CONTROL may provide facilities for interfacing with hardware, in a similar way to Fran's treatment of mouse and keyboard events. Unlike Fran, the event condition, given by a boolean behaviour, describes the event completely, and there is no value (the 'after' behaviour) packaged up with the condition.

A reactive behaviour is defined in terms of three behaviours: the first behaviour is used initially; the second is a boolean valued behaviour, specifying the event condition; the third is a new behaviour that is switched to as soon as the condition becomes true. For example, we could use a reactive behaviour to describe the output of a thermostatically controlled heater; it emits heat until the temperature reaches the desired level, at which point it switches to `lift0 0`, that is, off. Without a primitive operation for reactivity we would not be able to express this behaviour because there is no way of evaluating behaviours at particular times within the language, and so we could not determine when the temperature reaches the desired level.

The event condition for this example requires a lifted greater than or equal to function. The function

$$\texttt{>=} : \texttt{Real -> Real -> Bool}$$

has two arguments, so it can be lifted using `lift2` to obtain

$$\texttt{>=*} \;\equiv\; \texttt{lift2 (>=)} : \texttt{Beh Real -> Beh Real -> Beh Bool}.$$

Then the event condition that the behaviour `temp` : `Beh Real` has reached the level t_1 : `Real` is given by

$$\texttt{temp >=* lift0 } t_1.$$

Here we have used `>=*` as an infix operator for readability.

In general a reactive behaviour has three parts, which are as follows:

B is the *before* behaviour:	use this behaviour initially
C is the *condition* behaviour:	test this boolean behaviour to determine when it becomes true
D is the *after* behaviour:	when C becomes true, switch to this behaviour, and use it from now on.

A syntax with a natural reading for this is,

$$B \;\texttt{until}\; C \;\texttt{then}\; D. \tag{5.6}$$

The behaviours B and D must have the same type for the overall expression to make sense.

Here are a couple of examples to clarify the intended semantics. The following behaviour yields the value 1 until the time is 1 and then switches to 2:

```
(lift0 1) until (time >=* lift0 1) then (lift0 2).
```

It yields the value 2 for all times at or after 1. To emphasise that reactive behaviours switch permanently when the condition becomes true, the following behaviour is semantically identical to the previous one:

```
(lift0 1) until (time ==* lift0 1) then (lift0 2).
```

(`==*` is the lifted equality function.) The condition is only true for an instant when the time is 1, and is false for all times after 1, but the behaviour continues to yield 2 for all times after 1 because once it has reacted it switches to the after-behaviour permanently.

One alternative semantics is: yield B when C is false and D when C is true; that is, switch between B and D every time the value of C changes (this is equivalent to `lift3 if` for a conditional function `if` which takes three arguments). Practically this is not as useful as the operator described above because in most systems the occurrence of an event marks a change in the state of the system, and such changes are permanent—the event cannot "unoccur" regardless of the value of the condition that described the event. In other words, once an event has occurred the system responds in some way and then continues in a new state. Another important factor in favour of permanent switching is that it allows us to delete the behaviour B after the event has occurred, rather than keep it running in just case we need to switch back to it later. In functional programming terminology, the garbage collector can reclaim B after the event has occurred.

We will now formalise the semantics of `until-then`. To simplify the discussion we will assume for now that $[\![C]\!]$ does not map any times to bottom. Then it is a predicate, and therefore we can define the set of times when it is true,

$$T = \{t \in \mathbb{T} \mid [\![C]\!]t\}.$$

A general `until-then` expression of the form (5.6) should use B for any time t that is strictly before all the times in T, and otherwise it should use D. In other words, if t is not in the upperset of T, then use B, and otherwise use D. Note that T does not necessarily have a minimum element, so reactive behaviours do not always have an event time when they should switch from B to D. This point is quite subtle and is discussed in depth in Section 5.7.

The definition of uppersets is as follows:

Definition The upperset of $S \subseteq \mathbb{R}$ is given by

$$\uparrow S = \{s \in \mathbb{R} \mid \exists s' \in S : s' \leq s.\} \qquad \Box$$

A preliminary semantics for `until-then`, which ignores the possibility that

$[\![C]\!]$ may yield $\perp_{\mathbb{B}}$ for some times, is as follows:

$$[\![B \texttt{ until } C \texttt{ then } D]\!] \quad = \quad t \mapsto \left\{ \begin{array}{l|l} [\![B]\!]t & t \notin \uparrow T \\ [\![D]\!]t & \end{array} \right.$$
$$\text{where } T = \{t \in \mathbb{T} \mid [\![C]\!]t\}.$$

In general the situation is more complicated because for any time t, the value of $[\![C]\!]t$ may be $true$, $false$, or $\perp_{\mathbb{B}}$. This means that $[\![C]\!]$ is not simply a predicate as we assumed it is in the set comprehension for T above.

In order to determine whether to use B or D it is not necessary to know the value of $[\![C]\!]$ at all times—if it is only undefined (i.e., yields $\perp_{\mathbb{B}}$) at times after the reactive behaviour has switched then it makes no difference. More precisely, if the condition starts as false, becomes true later and subsequently becomes undefined, then we know exactly when to switch from B to D, despite the undefined points. This is not true if the condition is undefined before it is true, because then we do not know when we should switch. Therefore, in such cases the reactive behaviour is undefined from the time when the condition becomes undefined (it is B before this time).

In short, once the condition has yielded bottom it is no longer valid for determining when the event occurs, and once it has yielded true the event occurrence is known and subsequent values of the condition are irrelevant. Thus, given a time t we must consider three cases: if the condition has only ever been false, then use B; if the condition has, before or at time t, been true, and there are no bottoms before this true, then use D; otherwise there must be a bottom before a true and so the result is bottom. This suggests the following formal definition:

$$[\![B \texttt{ until } C \texttt{ then } D]\!] = t \mapsto \left\{ \begin{array}{l|l} [\![B]\!]t & t \notin \uparrow T \cup \uparrow Bad \\ [\![D]\!]t & t \in \uparrow T \wedge \uparrow T \not\supseteq \uparrow Bad \\ \perp & \end{array} \right. \tag{5.7}$$

where

$$\begin{array}{lcl} T & = & \{t \in \mathbb{T} \mid [\![C]\!]t = true\} \\ Bad & = & \{t \in \mathbb{T} \mid [\![C]\!]t = \bot_{\mathbb{B}}\}. \end{array}$$

5.3 Examples of reactivity

Figure 5.1 shows the values of four `until-then` expressions using the semantics given in Equation 5.7. Notice that we have used numbers, booleans, and the functions `>=` and `cos`, as if they were the lifted versions. We call this implicit lifting, and it is justified because it is always clear from context whether a constant refers to the usual value or the behaviour version. For example, if `1` is the first argument of an `until-then` expression, then it must mean `lift0 1` because `until-then` takes a behaviour as its first argument. Implicit lifting is not part of the language, but we use it in this dissertation to avoid an excessive proliferation of the lifting operators. In practice it can be implemented using overloading (Fran makes use of Haskell's overloading facilities to do this).

The first example is a straightforward application of the semantics so far. It is obtained by using the definitions for `until-then`, `lift0`, `lift2` and `time`. The second example shows that if the condition is always false then the `until-then` expression is equivalent to B. In other words, it proves the axiom

$$B \texttt{ until false then } D = B$$

We discuss other axioms involving behaviour expressions in Section 5.10. The third example is similar to the second except that the condition is always true, and so the `until-then` expression is equivalent to D. The fourth example is again a straightforward application of the definitions.

$$
\begin{aligned}
&[\![\texttt{1 until (time >= 1) then 2}]\!]\\
&\quad = \; t \mapsto \left\{ \begin{array}{c|l} 1 & t \notin \uparrow\{t \in \mathbb{T} \mid t \geq 1\} \\ 2 & \end{array} \right.\\
&\quad = \; t \mapsto \left\{ \begin{array}{c|l} 1 & t < 1 \\ 2 & \end{array} \right.
\end{aligned}
$$

$$
\begin{aligned}
&[\![B \; \texttt{until false then} \; D]\!]\\
&\quad = \; t \mapsto \left\{ \begin{array}{c|l} [\![B]\!]t & t \notin \uparrow\{t \in \mathbb{T} \mid false\}(= \emptyset) \\ [\![D]\!]t & \end{array} \right.\\
&\quad = \; t \mapsto [\![B]\!]t\\
&\quad = \; [\![B]\!]
\end{aligned}
$$

$$
\begin{aligned}
&[\![B \; \texttt{until true then} \; D]\!]\\
&\quad = \; t \mapsto \left\{ \begin{array}{c|l} [\![B]\!]t & t \notin \uparrow\{t \in \mathbb{T} \mid true\}(= \mathbb{T}) \\ [\![D]\!]t & \end{array} \right.\\
&\quad = \; t \mapsto [\![D]\!]t\\
&\quad = \; [\![D]\!]
\end{aligned}
$$

$$
\begin{aligned}
&[\![B \; \texttt{until (time >= cos time) then} \; D]\!]\\
&\quad = \; t \mapsto \left\{ \begin{array}{c|l} [\![B]\!]t & t \notin \uparrow\{t \in \mathbb{T} \mid t \geq cos(t)\} \\ [\![D]\!]t & \end{array} \right.
\end{aligned}
$$

Figure 5.1: Examples of applying the semantics of `until-then`

5.4 Implicit verses explicit values

Notice that we have not simplified the final example in Figure 5.1 as we have done for the other examples. To do so would require solving $t = cos(t)$ for the smallest $t \in \mathbb{T}$; that is, suppose t_1 is such a solution, then the condition simplifies to

$$t \notin \uparrow \{t \in \mathbb{T} \mid t \geq t_1\}) = t < t_1.$$

However, we cannot express this solution explicitly, that is, as a *formula*, because no formula exists. This does not mean that a solution does not exist—we can prove that it does—but it means that the only way we can express this value is as the solution to an equation.

This illustrates a peculiarity of our semantics: the semantics goes as far as providing implicit formulas (i.e., equations) for the mathematical denotations of programs, but sometimes some mathematical analysis is necessary in order to obtain explicit values. Furthermore, in cases such as the example above, it is not possible to express the value explicitly.

5.5 Nested `until-then` expressions

So far we have introduced the following four operators for constructing behaviour expressions:

`time, lift0, $*, until-then.`

The semantics of these operators has been defined compositionally, that is, in terms of the semantics of their arguments. Of course, the behaviours in an `until-then` expression could themselves be `until-then` expressions, and this raises some new questions regarding the semantics of reactive behaviours. We will explore these in this section.

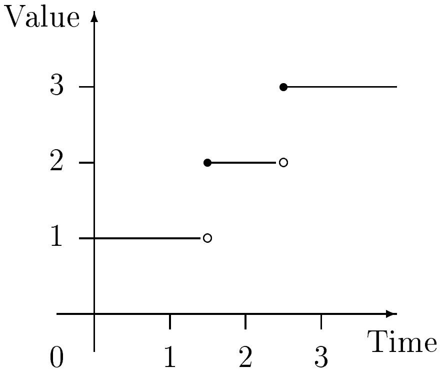
We are concerned with the interpretation of reactive behaviours of the form

$$B \texttt{ until } C \texttt{ then } D$$

where any/all of B, C, and D are themselves reactive behaviours. Let us consider the case when D is reactive to begin with. As an example, consider the nested expression,

$$\texttt{1 until (time >= 1.5) then}$$
$$\underbrace{\texttt{(2 until (time >= 2.5) then 3)}}_{D}$$

This behaviour should start as the constant behaviour $t \mapsto 1$, and then switch to D at time 1.5. Then it should be the constant behaviour $t \mapsto 2$ until time 2.5 when it should switch to 3; the graph of this behaviour is

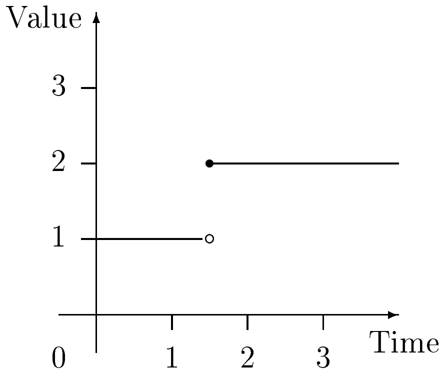


Fortunately, this is exactly the interpretation that our semantics gives, as can be verified by routine calculation.

Now consider a slight variation on the previous example which is the same except for the second condition, labelled C_2,

$$\texttt{1 until } \overbrace{\texttt{(time >= 1.5)}}^{C_1} \texttt{ then}$$
$$\underbrace{\texttt{(2 until } \overbrace{\texttt{(time <= 0.5)}}^{C_2} \texttt{ then } \overbrace{\texttt{ 3 }}^{D_2}\texttt{)}}_{D_1}.$$

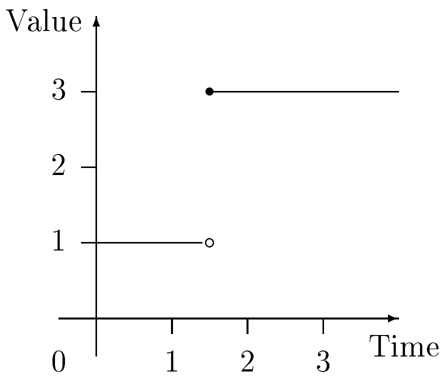
Intuitively we expect this behaviour to start as $t \mapsto 1$ and then switch to D_1 at time 1.5 as before. Then it should be $t \mapsto 2$ forever because the condition C_2 will never become true, since we have already passed time 0.5;



But this is not what our semantics gives, because it evaluates conditions over all times. The set of times when the second condition C_2 is true is given by

$$\begin{aligned} T_2 &= \{t \in \mathbb{T} \mid [\![C_2]\!]t\} \\ &= \{t \in \mathbb{T} \mid t \leq 0.5\} \\ &= [0, 0.5], \end{aligned}$$

and so when we switch to D_1 (at time 1.5) we will immediately switch to D_2 (i.e., the value 3) because all times are in $\uparrow T_2 = [0, \infty) = \mathbb{T}$; this results in the behaviour with graph



This is not what we intended. We want behaviours to be memoryless, that is, have an intrinsic meaning not dependent on what has gone before. From a practical perspective this property is essential; if it were absent it would be necessary to evaluate condition behaviours for all times in the past, which would be very inefficient. More importantly, it is not useful for conditions in nested reactive behaviours to apply for times in the past.

In summary, we should not evaluate conditions like C_2 since time began, but rather from the time when their enclosing behaviour (in this case, D_1) was switched to. In this example, we should test the condition C_2 from time 1.5 onwards. This is simple to capture semantically: the behaviour D_1 is used for all times after C_1 became true, that is, all times in the set,

$$\begin{aligned} &\uparrow T_1 \\ &\text{where } T_1 = \{t \in \mathbb{T} \mid [\![C_1]\!]t\} \end{aligned}$$

So we should only evaluate the condition C_2 at times in this set, which can be expressed as,

$$T_2 = \{t \in \uparrow T_1 \mid [\![C_2]\!]t\}$$

Thus all conditions should be evaluated with respect to some set of times which is the set of times when the behaviour is "alive"—all the times after the (enclosing) behaviour was switched to. This is contextual information necessary to make sense of conditions in reactive behaviours.

This contextual information must be passed through by our semantic function so that it is available to the components of compound expressions. Therefore $[\![\_]\!]$ needs an extra argument that gives the set of times when the behaviour is alive. (Recall that these sets do not always have a least time, so we cannot pass a single time denoting when the behaviour is alive from—see

$$
\begin{array}{rcl}
[\![\texttt{time}]\!](T_0) & = & t \mapsto t \\[2ex]
[\![\texttt{lift0}\ x]\!](T_0) & = & t \mapsto [\![x]\!] \\[2ex]
[\![fb\ \texttt{\$*}\ x]\!](T_0) & = & t \mapsto ([\![fb]\!](T_0)\,t)\ ([\![x]\!](T_0)\,t)
\end{array}
$$

$$
\begin{array}{l}
[\![B\ \texttt{until}\ C\ \texttt{then}\ D]\!](T_0) \\
\qquad\qquad = \ t \mapsto \left\{ \begin{array}{l|l} [\![B]\!](T_0)t & t \notin \uparrow T \cup \uparrow Bad \\ [\![D]\!](\uparrow T)t & t \in \uparrow T \wedge \uparrow T \not\supseteq \uparrow Bad \\ \bot & \end{array} \right.
\end{array}
$$

$$
\begin{array}{rrcl}
\text{where} & T & = & \{t \in T_0 \mid [\![C]\!](T_0)t = true\} \\
 & Bad & = & \{t \in T_0 \mid [\![C]\!](T_0)t = \bot_{\mathbb{B}}\}
\end{array}
$$

Figure 5.2: The semantic function $[\![\_]\!]$

Section 5.7.) The new definition of $[\![\_]\!]$ which includes this is given in Figure 5.2. Note that for `$*` the set T_0 is just passed through to its components, but for `until-then` the behaviours B and C are passed T_0, while the after behaviour D is passed $\uparrow T$, which is the times when it is alive.

We have covered the case when D is reactive, so now we need to consider the case when B or C are reactive. In fact, the new semantic function is correct for these cases because the behaviours B and C are alive as soon as the overall `until-then` statement is, that is, for times in the set T_0 in the semantic equation.

5.6 Integrals

Sometimes it is easier to describe the rate of change of a quantity than the quantity itself. It is essentially this observation that led Newton to develop the differential calculus. For example, it is easy to describe the accelerations (the rate of rate of change) of three bodies under gravitational attraction—each acceleration is proportional to the gravitational forces acting on the body—but it is very difficult to give a formula for their positions (in general it is impossible). To allow quantities to be described by their rate of change, CONTROL provides an operator that yields the integral of any given behaviour.

The integral of a top-level behaviour, f, is represented by `integral` f, and results in the behaviour that, at time t, yields the integral of $[\![f]\!]$ from 0 to t (i.e., the area under the graph of $[\![f]\!]$ from 0 to t). If `integral` is used in the after part of some reactive behaviour, then the integrand is integrated from the infimum of the times when it is alive, that is from $\inf(T_0)$. This appears to be at odds with `until-then` which distinguishes between events such as `time >= 1` and `time > 1`, as explained in Section 5.2. Taking the infimum of the set of times when a behaviour is alive yields the same for both of these behaviours, so `integral` does not make the distinction that `until-then` does. However, the reason we define integral this way is that including or excluding the endpoint of the interval integrated over does not change its value—a line of zero width has no area. So taking the infimum is the simplest way to define `integral` using the standard notation for definite integrals.

The formal definition is as follows:

$$[\![\texttt{integral}\ f]\!](T_0) \quad = \quad t \mapsto \int_{\inf(T_0)}^{t} ([\![f]\!](T_0)\, s).ds$$

Integration only makes sense for real-valued behaviours, so f : `Beh Real`. The integrand $[\![f]\!]$ may have undefined points, that is, map certain times to $\perp_{\mathbb{R}}$. Such points could make the integral undefined; for example, if the undefined points are singularities (undefined values resulting from a division by zero). If the integral is undefined, the above integral expression should yield $\perp_{\mathbb{R}}$.

We have not said how we should evaluate the integral expression in the above definition. This must be done using mathematical analysis, and need not concern us. Our semantics gives an implicit description of the meaning of any given behaviour in terms of equations and integrals. If we require an explicit formula for the function of time that the behaviour represents, then it is obviously necessary to do some mathematical analysis. Our theory accounts for the possibility that there may not be a unique solution to the equations by allowing the result to be bottom. In practice, however, it is sometimes not possible to solve the equations even when solutions exist, and this limits our ability to reason about such programs. This is a consequence of the limitations of current mathematical knowledge, however, and is not due to our approach to giving a semantics to CONTROL.

A related issue is the meaning of integrals of reactive behaviours. Reactive behaviours are particular to CONTROL, and so we must define what it means to integrate such behaviours. This is straightforward because our semantics interprets behaviours as functions of time, and reactive behaviours are just piecewise functions of time. Integrating piecewise functions is well understood; integrate the pieces and add up the results. In summary, the definition of `integral` is well defined for all real valued behaviours.

5.7 Avenue on event times

The usual notion of an event is some occurrence which happens at a particular time; for example, two objects colliding or a temperature reaching a given level. This is the view taken by Arctic, Fran and most of the discrete time languages we have seen. In languages using continuous time where it is possible to specify events by boolean behaviours, this view must assign an event time to events such as $t > 1$ which have no earliest time when they are true. An event condition using greater than may correspond to the event that an object has passed a given position, or that a temperature has exceeded a given level. The distinction is quite subtle; using $>$ instead of $\geq$ in a condition behaviour only makes a difference at one value, and so it makes little difference in languages using approximation techniques. But our theory is exact, and so the difference is vitally important. In this section we will consider the different approaches in Arctic, Fran and CONTROL with regard to this issue.

The simplest example of a condition that has no earliest time when it becomes true is the behaviour

$$\texttt{time > 1}.$$

This represents the function

$$c(t) = t > 1$$

which is true for times in the set

$$T = \{t \in \mathbb{T} \mid c(t)\} = (1, \infty).$$

This set has no minimum element and therefore there is no earliest time when the condition becomes true. If all events must have an occurrence time, then we must have some way of calculating the event time from the set T. Notice that the semantics we gave to `until-then` in CONTROL does not have to address this issue because it switches to the new behaviour for all times in the upperset of T, and uppersets exist for any set. Thus the behaviour

$$B\ \texttt{until time > 1 then}\ D$$

will act like B for times in the set $[0,1]$ and like D for times in the set $(1,\infty)$.

Arctic simply ignores the problem. The last paragraph in Section 8 of [Dan84] states that

> ... a boolean function is evaluated to find the first moment [after it came alive] at which the function is true.

This semantics cannot be applied to conditions like $t > 1$ because there is no first moment when it is true.

Fran recognises the problem, and avoids it by taking the event time to be the infimum of times when the condition is true. Infimums of sets of real numbers always exist (see [Apo74]) so there is always an event time if the condition is ever true. Fran's time domain is extended with ∞ so that if the event condition never becomes true the event time is ∞; this simply requires the infimum of the empty set to be ∞, which causes no difficulty.

Although Fran assigns an event time to any possible condition behaviour, it is not as refined as CONTROL because events like $t \geq 1$ and $t > 1$ are semantically the same in Fran—they both have an event time of 1. A reactive behaviour in Fran switches strictly after its event time, so for both conditions $t \geq 1$ and $t > 1$ the old behaviour is used at time 1 and the new one strictly after time 1. If Fran switched to the new behaviour at time 1, it would seem to early for the event $t > 1$ because it would switch before the event condition has ever been true. On the other hand, switching strictly after time 1, as Fran does, seems to late for the event $t \geq 1$ which is true at time 1. As we shall see in the next section, the only reactivity construct that yields the old behaviour for times when the event condition has never been true and the new behaviour otherwise is the one we have defined for CONTROL.

We suspect that the reason reactive behaviours in Fran switch strictly after the event time is that this is what the implementation does. In the implementation this avoids recursive reactive behaviours looping when sampled, and it may have been hoped that this method which works in discrete time also carries over to continuous time. We have found this not to be the case, as we shall see in Section 6.11.

5.8 Avenue on alternative semantics for reactivity

In this section we will consider alternative semantics that could be given to `until-then`, and whether any semantics other than CONTROL's or Fran's is reasonable. Firstly we must define what we regard as a reasonable semantics for `until-then`. Given that our language is idealised, we expect reactive behaviours to respond to events without any delay; otherwise the language would be approximate, and our approach is to avoid the complexity of approximation by first considering the exact language.

We will consider the two event conditions `time >= 1` and `time > 1`, which represent the values $t \geq 1$ and $t > 1$, and the semantics of reactive behaviours using these event conditions. All conditions are equivalent to one of these two in the sense that they either become true at a particular time, or for times strictly after some time.

Firstly, a reactive behaviour with the condition $t \geq 1$ only has two choices; either switch at time 1 or strictly after. Switching any finite length of time before or after time 1 would clearly be an approximate response. These two choices only differ at time 1—for all other times they are the same.

Similarly, a reactive behaviour with the condition $t > 1$ could switch either at time 1 or strictly after. So there are two choices for both kinds of event, giving four possible semantics for `until-then` that are not approximate. It would be absurd to switch at time 1 for the condition $t > 1$ and

	$t \geq 1$	$t > 1$
CONTROL	at 1	after 1
Fran	after 1	after 1
Early	at 1	at 1

Figure 5.3: Different semantics of `until-then`

strictly after time 1 for the condition $t \geq 1$, however, so there are three possibilities that are reasonable. These are shown in Figure 5.3. The first corresponds to the semantics of `until-then` in CONTROL, the second to the semantics in Fran and the third we have called Early because it switches at time 1 in both cases.

In summary, there are three different semantics for `until-then` that are reasonable in the sense that they do not switch a finite duration before or after the first time or times when the event condition becomes true. CONTROL's semantics gives the before-behaviour (in a reactive behaviour) for times before any time when the condition is true and the after-behaviour otherwise. The other semantics can be defined by finding event times by taking the infimum of the times when the condition is true, and then either switching strictly after that time (Fran) or at that time (Early). At this stage it seems that CONTROL's semantics is the most natural, and it is more refined because it distinguishes between events like $t \geq 1$ and $t > 1$ which the other semantics do not, but the other semantics are still reasonable possibilities. Later on in Section 6.11 we discuss how the choice affects the semantics of recursive reactive definitions.

5.9 Avenue on integrability

Any real valued behaviour can be used as an argument to the `integral` operator yielding the integral if the behaviour is integrable and bottom if it is not. In this section we consider criteria for classifying behaviours according to integrability.

Firstly we need a precise definition of integration. The standard definition uses Riemann sums which are approximations of the area under the curve obtained by dividing the interval into strips and making a rectangle the height of the curve at some point in each strip. The sum of the area of these rectangles is the Riemann sum, and if the limit as the width of the strips tends to zero converges, then this is the value of the integral. If it does not then the function is not integrable over that interval. (The limit has to converge regardless of what point in each strip is chosen for the height of the rectangle.) The formal definition of Riemann-integrability and other technical terms in this section can be found in [Apo74].

All continuous functions are Riemann-integrable (from now on, integrable). We use continuous in at least three different ways in this dissertation, but when we are referring to real valued functions we mean continuous in the sense of real analysis and not domain theory.

Many behaviours do not represent continuous functions, for example:

1. `lift1 floor time`
2. `1 / (time - 1)`
3. `1 until (time >= 1) then 2`

The first behaviour lifts the `floor` function which is discontinuous. The second has a discontinuity because of division by zero at time 1. The third explicitly defines a step function using a reactive behaviour. The first and third examples are integrable, and more generally all bounded functions with discontinuities at discrete points are integrable (see [TF92]). A stronger result is the following:

THEOREM 5.1 (LEBEGUE'S CRITERIA FOR RIEMANN-INTEGRABILITY) *If f is bounded on $[a, b]$ and the set of discontinuities S on $[a, b]$ has zero measure, then f is Riemann-integrable on $[a, b]$.*

Sets with zero measure include all countable sets as well as some peculiar uncountable sets (such as the Cantor set, [Apo74, pp. 180]). A reactive behaviour may react many times, but event occurrences are in sequence and so the set of discontinuities in any reactive behaviour must be countable. Therefore it follows that any bounded reactive behaviour is integrable (assuming that no built in functions that are not integrable are lifted).

5.10 Avenue on axioms

There are a few simple axioms which hold for behaviour expressions. Using our semantics it is straightforward to verify these axioms. In Section 5.3 we saw the following equivalences:

$$B \texttt{ until false then } D = B$$

$$B \texttt{ until true then } D = D$$

The following property of lifting holds:

```
(lift0 f) $* (lift0 a)  =  lift0 (f a)
```

Assuming a function `integrate` which calculates the symbolic integral of behaviours, we have

```
integral b  =  integrate b
```

Often the symbolic integral does not exist or is difficult to compute, so this equivalence only applies to a relatively small subset of behaviours.

There are very few useful axioms in terms of the basic operators. In particular, `until-then` is not associative.

Chapter summary

There are four operators for constructing behaviour expressions: `lift0`, `$*`, `until-then` and `integral`. The lifting operators provide a way of applying existing functions to behaviours. The `until-then` operator is used for expressing reactivity. It is quite subtle for two reasons:

1. Conditions specifying events do not always have a first time when they occur.

2. Nested `until-then` expressions must test conditions from when their enclosing behaviour came alive, and not for all times.

These considerations lead to defining event occurrences in terms of uppersets of times when the condition is true, and defining a semantic function that passes these uppersets on to after-behaviours.

The `integral` operator is relatively straightforward because reactive behaviours can be integrated in a piecewise fashion. However, we can not always obtain explicit formulas for the meaning of `integral` expressions as the existing techniques for analytical integration do not cover all cases.

Chapter 6

Behaviour definitions

So far we have seen how to write behaviour expressions which are used to represent functions of time. In this chapter, we will introduce behaviour definitions which let us name behaviours by variables and then refer to those behaviours elsewhere by their names. This can be useful when a behaviour expression appears more than once in a program, because it avoids duplicating the expression. More importantly, if behaviour definitions are allowed to be recursive then they increase the expressiveness of the language considerably. In this chapter we discuss simple recursive definitions and later on in Chapter 9 we give realistic examples of programs that cannot be written in CONTROL without using them.

As we shall see, the standard approach for giving a semantics to recursive definitions does not work for behaviours. This leads us to develop a new approach, which is the main subject of this chapter.

6.1 Recursive behaviour definitions

A common syntax for defining multiple, mutually recursive functions is,

$$\begin{array}{rcl} \texttt{letrec}\ a_1 & \texttt{=} & E_1 \\ & \vdots & \\ a_n & \texttt{=} & E_n \\ \texttt{in}\ F & & \end{array}$$

where each variable a_i may appear in any E_j, and, of course, in the body F. It is much simpler to describe the semantics of a single recursive definition, and in fact it is sufficient to do so because there is a standard method of translating multiple recursive definitions into a single nested recursive definition (see Section 7.7).

We will call our construct for recursive definitions `letbeh` to avoid confusion with `letrec`, and to emphasise that it can only be used to define behaviours. The syntax is,

$$\texttt{letbeh}\ a\ \texttt{=}\ E\ \texttt{in}\ F$$

where a is a variable, and E and F are behaviours.

Many recursive behaviour definitions, such as

`letbeh a = a in a`,

are not meaningful, but are two situations where they are particularly useful:

1. Defining a behaviour that changes when it reaches a certain value. This requires a reactive behaviour where the condition refers to the behaviour being defined.

2. Defining a behaviour in terms of its rate of change using `integral`, where the rate of change refers to the behaviour being defined. This corresponds to integral equations in mathematics, which are essentially ordinary differential equations expressed differently.

Recursive reactive definitions turn out to be quite difficult to give a semantics to because the standard approach does not work. In the following sections we will explain the problem with applying the standard approach, and the method by which we give a semantics to these definitions. Finally, we will extend the method to include recursive integral definitions.

6.2 Recursive reactive definitions

To describe a behaviour that changes when it reaches some value, a recursive definition of the following form is required:

$$\texttt{letbeh}\ a\ \texttt{=}\ \ldots\ B\ \texttt{until}\ C[a]\ \texttt{then}\ D\ \ldots\ \texttt{in}\ F. \tag{6.1}$$

That is, part of the expression defining a is reactive and changes when a reaches some value, as specified by the condition $C[a]$ that depends on a. So these kinds of behaviours are ones that react to themselves. If we consider multiple definitions like these, where the conditions refer to any of the variables, we see that we are describing behaviours that interact with each other. Interaction between components is an essential aspect of most reactive systems, and in CONTROL it is not possible to express interaction without this kind of recursive definition.

We will now consider a subset of definitions of the form (6.1), where the right hand side is an `until-then` expression at the top-level,

$$\texttt{letbeh}\ a\ \texttt{=}\ B\ \texttt{until}\ C[a]\ \texttt{then}\ D\ \texttt{in}\ F. \tag{6.2}$$

We restrict our attention to this class of definitions because they have a simple meaning; they are equivalent to non-recursive definitions of the form,

$$\texttt{letbeh}\ a\ \texttt{=}\ B\ \texttt{until}\ C[B]\ \texttt{then}\ D\ \texttt{in}\ F.$$

The condition $C[a]$ is only relevant before the event occurs—afterwards a is D and $C[a]$ is no longer required—and before the event occurs a acts like B. This is a consequence of a basic causality requirement; D will be used after the event has occurred, and so it should not affect the event itself. This principle is not adhered to if we take a naive approach to recursion, however, because the semantics of `until-then` stipulates that as soon as the condition becomes true, the behaviour switches to some new behaviour, and in programs like (6.2) this means that D is not used only *after* the event, it is used *at* the event time as well. We will now describe this problem formally.

6.3 Least fixed points

To give a compositional semantics to a language with definitions, we need some way of capturing bound variables so that we can interpret them in sub-expressions where they appear free. This can be achieved by passing an environment to the semantic function; then the environment gives the values of the free variables in every program phrase. Abstractly, an environment, u, is a function from variables to values, and we write $[u|a : x]$ for the environment that is identical to u except that it maps a to x, overriding any previous assignment for a in u. We write $[\![P]\!]u$ for the meaning of the program P in the environment u.

Using environments, the semantics of `letrec` prescribes that a recursive definition means a solution to the corresponding equation in the appropriate semantic domain; that is,

$$\begin{aligned} [\![\texttt{letrec } a \texttt{ = } E \texttt{ in } F]\!]u &= [\![F]\!][u|a:x] \\ x &= [\![E]\!][u|a:x]. \end{aligned}$$

To re-iterate, the meaning of a is a solution to the equation in x. For a PCF-like language, these equations can be solved by expressing the problem slightly differently, as finding the fixed points (i.e., x_1 such that $G(x_1) = x_1$) of the function

$$G(x) = [\![\mathtt{E}]\!][u|a:x],$$

and then choosing the least fixed point with respect to an information ordering as discussed in Section 4.2. This ordering is such that there is always a least solution (so we have a canonical choice for the meaning of a), and that this is the solution we require from a computational perspective.

Adopting this approach for recursive behaviour definitions yields,

$$\begin{aligned} [\![\texttt{letbeh } a \texttt{ = } E \texttt{ in } F]\!](T_0)u &= [\![F]\!](T_0)[u|a:x] & (6.3) \\ x &= [\![E]\!](T_0)[u|a:x] & (6.4) \end{aligned}$$

(i.e., as for `letrec` but with the extra argument T_0 which is the set of times when the behaviour argument is alive). For now, we will ignore the issue of choosing canonical solutions to these equations, and just consider whether solutions exist.

Let E in Equations (6.3) and (6.4) be

$$E \equiv B \texttt{ until } C[a] \texttt{ then } D.$$

Then Equation (6.4) is

$$x = [\![B \texttt{ until } C[a] \texttt{ then } D]\!](T_0)[u|a:x].$$

Now, using the semantics of `until-then` given in Figure 5.2 (we assume that $C[a]$ is not bottom in this discussion) we obtain,

$$\begin{aligned} x &= t \mapsto \left\{ \begin{array}{l|l} [\![B]\!](T_0)[u|a:x]t & t \notin \uparrow T \\ [\![D]\!](\uparrow T)[u|a:x]t & \end{array} \right. \\ T &= \{t \in T_0 \mid [\![C[a]]\!](T_0)[u|a:x]t\}. \end{aligned}$$

At this stage it is instructive to try this semantics with some programs; for example, applying the semantics to the program

```
letbeh a = 1 until (time >= a) then 2 in a.
```

Doing so yields the following equation for x (we let $T_0 = \mathbb{T}$):

$$x = t \mapsto \left\{ \begin{array}{l|l} 1 & t \notin \uparrow T \\ 2 & \end{array} \right. \tag{6.5}$$

$$T = \{t \in \mathbb{T} \mid t \geq x(t)\}. \tag{6.6}$$

There are no solutions to these equations, which we will now prove formally.

PROPOSITION 6.1 *Equations (6.5) and (6.6) have no solutions for $x \in \mathbb{T} \to \mathbb{R}_\perp$ and $T \in \mathbb{P}(\mathbb{T})$.*

PROOF For $t < 1$ the condition $t \geq x(t)$ must be false because $x(t)$ is either 1 or 2. Therefore, T contains no times less than 1, and so $t \notin \uparrow T$ is true for $t < 1$. Hence, by (6.5) we have $x(t) = 1$ for $t < 1$. But at $t = 1$ there is a kind of Russellian paradox. We know that $x(1)$ is either 1 or 2. Let us consider both cases:

- Suppose that $x(1) = 1$. Then $1 \geq x(1) \implies 1 \in T$ by (6.6), and so $1 \in \uparrow T$. But $1 \in \uparrow T \implies x(1) = 2$ by (6.5), which contradicts our assumption.

- Suppose that $x(1) = 2$. Then $1 \not\geq x(1) \implies 1 \notin T$ by (6.6), and, since T contains no times less than 1 or 1 itself, we have $1 \notin \uparrow T$. But $1 \notin \uparrow T \implies x(1) = 1$ by (6.5), which contradicts our assumption. □

In short, if we assume that $x(1) = 1$, then equations (6.5) and (6.6) imply that $x(1) = 2$, and conversely, if we assume that $x(1) = 2$, then equations (6.5) and (6.6) imply that $x(1) = 1$.

We could conclude from this that the meaning of the program is $\bot_{\mathbb{T}\to\mathbb{R}_\bot}$ (or is $t \mapsto 1$ for $t < 1$ and undefined for $t \geq 1$), but we want to give such programs a stronger meaning—in fact, they are only useful if we can do so.

The problem is exactly the same in the general case (6.2), and it arises because `until-then` stipulates that as soon as the condition becomes true it should yield the after-behaviour, D, but this changes the condition at the instant it becomes true, so it may then not be true (as in the example above). The only way to avoid this contradictory situation, and retain the view of recursive definitions as solutions to equations, is to delay switching slightly. This goes against our intention that CONTROL is an idealised, instantaneous response language, and moreover, it can be shown that all reasonable possibilities for defining `until-then` this way lead to unacceptable anomalies in the semantics. (These alternatives are explored in Section 6.11.) Therefore we must take a different approach to the semantics of recursive reactive definitions.

6.4 Non-reactive evaluation

In this section we will give an informal description of our solution to the problem with recursive reactive definitions. We have just seen a proof that some recursive reactive behaviours denote bottom because a contradiction arises at the times when the condition becomes true. Put simply, interpreting a definition of the form

$$a\ \texttt{=}\ B\ \texttt{until}\ C[a]\ \texttt{then}\ D,$$

leads to a contradiction because the condition $C[a]$ depends on the meaning of a, and a changes from B to D at the instant $C[a]$ becomes true, which means it may not be true after all.

We could avoid such contradictions by preventing reactive behaviours from switching when determining event occurrences. The idea is to interpret all reactive behaviours of the form

$$B \; \texttt{until} \; C \; \texttt{then} \; D$$

as if they were just B. We call this *non-reactive evaluation.* It is necessary to evaluate the whole program this way because any reactive behaviour, including nested `until-then`'s, could cause the problem. Note that before any events have occurred, interpreting programs using non-reactive evaluation is no different from an interpretation that takes reactivity into account.

As an example, the reactive behaviour

```
1 until (time >= a) then 2
```

is interpreted non-reactively as the behaviour `1`. This ignores the condition which refers to a variable, possibly recursively.

Say there are n `until-then` expressions in our program, and we refer to them by A_i, where,

$$A_i \equiv B_i \; \texttt{until} \; C_i \; \texttt{then} \; D_i.$$

One of these, say A_e for some index e, must react first (we will begin by ignoring the possibilities of no events ever occurring and of simultaneous events occurring). Using non-reactive evaluation, we can find the set of times, T_i, when C_i is true, for each `until-then` expression. For the behaviour that reacts first, A_e, the set T_e must contain earlier times than all the other sets T_i. This means that we know non-reactive evaluation will give the correct meaning of the program for times before those in T_e, because no event has occurred. It does not tell us anything about the value of any behaviours for times in $\uparrow T_e$, because they may depend on the value of the expression A_e, which has reacted (and non-reactive evaluation assumes it has not reacted).

We can evaluate behaviours for times in $\uparrow T_e$ if we replace A_e by D_e in our program, to account for it reacting. Then, if we use non-reactive evaluation again, we will get the correct meaning up to the next event occurrence. This suggests the following iterative procedure for interpreting behaviours in a program P:

1. Evaluate P non-reactively.
2. For each reactive behaviour A_i, find T_i.
3. Let T_e be the set with the earliest times.
4. The evaluation in 1 is valid for times before T_e.
5. To evaluate P for times in $\uparrow T_e$, replace A_e by D_e in P, and repeat this procedure.

6.5 Transitions

We will now formalise the procedure described above. The procedure yields a sequence of programs (P_i) beginning with the program $P(= P_0)$. Each program in the sequence is the same as the previous one except that one reactive behaviour (or more, if events are simultaneous)—say

$$A_{e_i} \equiv B_{e_i} \texttt{ until } C_{e_i} \texttt{ then } D_{e_i}$$

for the i-th program—has been replaced by D_{e_i} (this of course assumes that A_{e_i} is the first behaviour to react in P_i).

This is a kind of reduction, similar to term re-writing or reduction in the lambda calculus. One difference is that we are not only interested in the sequence of programs (P_i); we are also interested in the result of non-reactively evaluating each program and in the intervals when these evaluations are valid. Thus, we begin by evaluating $P : \theta$ non-reactively over $\mathbb{T}$. This yields $p_0 \in \mathbb{T} \to [\![\theta]\!]$, and enables us to find the set of times T_{e_0} when the first condition C_{e_0} becomes true. Then we know that p_0 is the meaning of P over the interval $\mathbb{T} \setminus \uparrow T_{e_0}$. We will annotate arrows denoting transitions with these values using the following notation:

$$P \xrightarrow[\mathbb{T} \setminus \uparrow T_{e_0}]{p_0} P_1.$$

For the second transition we evaluate P_1 over the times $\uparrow T_{e_0}$ to obtain the meaning p_1 which is valid over the interval $\uparrow T_{e_0} \setminus \uparrow T_{e_1}$; hence the second transition is

$$P_1 \xrightarrow[\uparrow T_{e_0} \setminus \uparrow T_{e_1}]{p_1} P_2,$$

and so on.

In general, we need to define a transition relation such that a program P is related to P' when evaluating P non-reactively over T_0 (the times when P is alive) yields p and this is valid up to times in T_1—in other words over the interval $T_0 \setminus T_1$. Using our notation this is written as

$$\mathtt{P} \xrightarrow[T_0 \setminus T_1]{p} \mathtt{P'}.$$

For example, the reactive behaviour

```
1 until (time >= 1) then 2
```

alive for times in $\mathbb{T}$ makes a transition to the behaviour `2`, and acts like the behaviour `1` over the interval $[0, 1)$, thus,

$$\mathtt{1} \xrightarrow[\mathbb{T} \setminus [1,\infty)]{t \mapsto 1} \mathtt{2}.$$

The behaviour `1` is shorthand for `lift0 1` which is non-reactive and therefore equals $t \mapsto 1$ over any interval.

This transition relation can be defined compositionally, that is, on the syntactic structure of programs. This way the transition P makes is calculated from the transitions of its immediate sub-phrases, and so on, producing a tree-structured derivation of the overall transition.

We must ensure that only the behaviour (or behaviours) that reacts first makes a transition. So, if C_e is the first condition to become true then at some place in the tree we will have

$$B_e \texttt{ until } C_e \texttt{ then } D_e \xrightarrow[T_0 \setminus \uparrow T_e]{b_e} D_e$$

and as we go down the tree this behaviour is combined with other behaviours in such a way that only this one changes. (The other behaviours react later and therefore should not change.)

As an example of a behaviour with two sub-phrases, we will consider A`+`B where `+` is the behaviour level addition operator. This is a special case of `lift2`, and in turn `lift2` is defined in terms of `lift0` and `$*`, but it is simpler to use `+` to illustrate transitions of compound expressions.

Say we take a bottom up approach to constructing the derivation tree for the transition A`+`B makes, and start by finding the transitions

$$A \xrightarrow[T_0 \setminus T_A]{a} A' \qquad B \xrightarrow[T_0 \setminus T_B]{b} B'. \tag{6.7}$$

This means that A is non-reactive for times before those in T_A, and similarly for B. The overall expression A`+`B is non-reactive over the smaller of the intervals $T_0 \setminus T_A$ and $T_0 \setminus T_B$. This gives three possibilities for the transition that A`+`B makes:

1. A reacts first ($T_A \supsetneq T_B$):

$$A \texttt{ + } B \xrightarrow[T_0 \setminus T_A]{t \mapsto a(t)+b(t)} A' \texttt{ + } B.$$

2. B reacts first ($T_B \supsetneq T_A$):

$$A \texttt{ + } B \xrightarrow[T_0 \setminus T_B]{t \mapsto a(t)+b(t)} A \texttt{ + } B'.$$

3. A and B react simultaneously ($T_A = T_B$):

$$A \texttt{ + } B \xrightarrow[T_0 \setminus T_A]{t \mapsto a(t)+b(t)} A' \texttt{ + } B'.$$

This bottom up approach works, but we can reduce the number of rules by taking a top down approach. All we need is a rule for Case 3, so A+B makes a transition to A'+B', and a no-change rule which allows B' to be B when Case 1 applies, or A' to be A when Case 2 applies. We will now explain this in more detail.

6.6 The no-change rule

Taking a top-down approach means that we start by trying to find the transition the overall behaviour makes. Continuing our last example, we want to find the transition that the behaviour A+B makes. It is non-reactive over the interval $T_0 \setminus M$, where M is either T_A or T_B, whichever contains the earliest times. Then we find the transitions that A and B make, as in 6.7. But we require the transitions they make over $T_0 \setminus M$ and so one of them may have to make a no-change transition; if A reacts first, B will have to make a no-change transition and vice-versa.

The rule for addition of behaviours, which we call lift2 (+), is

$$\frac{A \xrightarrow[T_0 \setminus M]{a} A' \quad B \xrightarrow[T_0 \setminus M]{b} B'}{A \texttt{ + } B \xrightarrow[T_0 \setminus M]{t \mapsto a(t)+b(t)} A' \texttt{ + } B'} \langle \text{lift2 } (+) \rangle$$

This captures Case 3 with $M = T_A = T_B$. The derivations for the premises are exactly as in 6.7.

If A reacts first (Case 1) then $T_A \supsetneq T_B$ (T_A contains earlier times than T_B) and B makes a no-change transition. Firstly we will describe the no-change rule, which is as follows:

$$\frac{B \xrightarrow[T_0 \setminus T_B]{b} B'}{B \xrightarrow[T_0 \setminus X]{b} B} \langle\text{no-change}\rangle \left(\begin{array}{c} X \supsetneq T_B \\ X = \uparrow X \end{array} \right)$$

The side condition $X \supsetneq T_B$ specifies that X contains earlier times than T_B (recall that T_B is an upperset, so if X contains more times than T_B then it must contain earlier times). Since X contains earlier times, the interval $T_0 \setminus X$ must be strictly smaller than $T_0 \setminus T_B$ and therefore B must make a no-change transition (i.e., a transition from B to B) over this smaller interval, as the rule dictates. The second side condition is necessary to ensure that X is an upperset, because otherwise $T_0 \setminus X$ would not be an interval.

The following derivation shows how the no-change rule and the lift2 (+) rule can be used to deal with Case 1:

$$\frac{A \xrightarrow[T_0 \setminus T_A]{a} A' \quad \dfrac{B \xrightarrow[T_0 \setminus T_B]{b} B'}{B \xrightarrow[T_0 \setminus T_A]{b} B} \langle\text{no-change}\rangle \left(\begin{array}{c} T_A \supsetneq T_B \\ T_A = \uparrow T_A \end{array} \right)}{A \texttt{ + } B \xrightarrow[T_0 \setminus T_A]{t \mapsto a(t)+b(t)} A' \texttt{ + } B} \langle\text{lift2 (+)}\rangle$$

Notes:

1. The transitions for A and B at the leaves (top) are exactly as in 6.7.

2. For Case 1, $T_A \supsetneq T_B$ is true.

3. $T_A = \uparrow T_A$ is true because T_A is an upperset.

The result of this derivation is that a transition to $A' + B$ is made, which is what we require for Case 1. Case 2 is symmetrical to this one.

A concern is that the choice of rules in derivations is no longer deterministic because the no-change rule can be used anywhere. This would be a problem if it resulted in many different meanings for some programs because we want our semantics to give a unique meaning to all programs. In fact this is not the case and it is easy to show that the rules are deterministic so long as the non-reactive interval for every transition is as long as possible; see Theorem 8.10 in Section 8.7.

The lift2 (+) rule above is a special case of the lift2 rule which in turn is a derived rule from the basic lift0 and $* rules. These two rules are straightforward because they yield the same values as in Figure 5.2. The lift0 rule is valid over the interval $T_0 \setminus \emptyset = T_0$ because the value the behaviour yields never changes. The set T_0 is an upperset and thus the value is valid for all times in the future. It is therefore irrelevant what behaviour this rule makes a transition to, and so the rule yields the empty term ε as the next behaviour;

$$\text{lift0} \qquad \frac{}{\texttt{lift0}\ x \xrightarrow[T_0 \setminus \emptyset]{t \mapsto [\![x]\!]} \varepsilon}$$

The rule for `$*` uses the same method as the lift2 rule to deal with the three cases when FB reacts first, A reacts first, or FB and A react simultaneously;

$$\$* \qquad \frac{FB \xrightarrow[T_0 \setminus M]{fb} FB' \quad A \xrightarrow[T_0 \setminus M]{a} A'}{FB\ \texttt{\$*}\ A \xrightarrow[T_0 \setminus M]{t \mapsto (fb(t))(a(t))} FB'\ \texttt{\$*}\ A'}$$

The rule for `time` is as we would expect:

$$\text{time} \qquad \frac{}{\texttt{time} \xrightarrow[T_0 \setminus \emptyset]{t \mapsto t} \varepsilon.}$$

6.7 Transitions for reactive behaviours

The most important transition rules are those for `until-then` because they allow a reactive behaviour to update when an event occurs, which is the purpose of the transition system. When an event occurs, part of the derivation tree for the transition will be of the form

$$\frac{B \xrightarrow[T_0 \setminus T_B]{b} B' \quad C \xrightarrow[T_0 \setminus T_C]{c} C'}{B \ \texttt{until} \ C \ \texttt{then} \ D \xrightarrow[T_0 \setminus \uparrow T]{b} D} \qquad \text{(ut-a)}$$

where $T = \{t \in T_0 \mid c(t)\}$ (this assumes that c is not bottom). This derivation is only valid when the event specified by C occurs before either B or C react, that is, when

$$\uparrow T \supsetneq T_B \cup T_C.$$

Otherwise (i.e., when B or C reacts first) the derivation is like that for lift2 (+); we introduce a variable M to represent the earlier of T_B and T_C and use the following derivation:

$$\frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \ \texttt{until} \ C \ \texttt{then} \ D \xrightarrow[T_0 \setminus M]{b} B' \ \texttt{until} \ C' \ \texttt{then} \ D.} \qquad \text{(ut-b)}$$

As for lift2 (+), this rule, in conjunction with the no-change rule, is sufficient to deal with the three cases (i.e., when B reacts first, when C reacts first, and when B and C react simultaneously). Notice that we are not concerned with when D reacts because it is not yet alive. The condition when this rule applies is when either B or C reacts before the condition specified by C becomes true, that is, when

$$M \supsetneq \uparrow T.$$

What about the case when C becomes true exactly when B or C react, that is, when $\uparrow T = M$? In this case we give priority to the top-level reactive behaviour, and make the transition to D as in (ut-a) above.

There is an alternative derivation for (ut-a) that achieves the same result and is more like (ut-b). The rule obtained is preferred because then the rules for the two cases are similar. All we need to do is replace T_B and T_C in (ut-a) with M which represents the earlier of the two uppersets, that is, $M = T_A \cup T_B$. Then the derivation is as in (ut-a) with possible applications of the no-change rule when T_A and T_B are not both equal to M. The side condition when this derivation is valid is $\uparrow T \supseteq M$ (recall that this rule applies when B or C react at or after the time/s when C becomes true, which is when the set $\uparrow T$ is a superset of M). Thus, the rule for derivations like (ut-a), which we call occ, is

$$\text{occ} \qquad \frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \texttt{ until } C \texttt{ then } D \xrightarrow[T_0 \setminus \uparrow T]{b} D} \; (\uparrow T \supseteq M)$$

An example of applying the occ rule is:

$$\frac{1 \xrightarrow[\mathbb{T} \setminus \emptyset]{t \mapsto 1} \varepsilon \quad time >= 1 \xrightarrow[\mathbb{T} \setminus \emptyset]{t \mapsto t \geq 1} \varepsilon}{\texttt{1 until (time >= 1) then 2} \xrightarrow[\mathbb{T} \setminus [1,\infty)]{t \mapsto 1} \texttt{2}} \qquad \text{(ex-occ)}$$

Here the set T is given by

$$\begin{aligned} T &= \{t \in \mathbb{T} \mid (t \mapsto t \geq 1)(t)\} \\ &= \{t \in \mathbb{T} \mid t \geq 1\} \\ &= [1, \infty). \end{aligned}$$

The rule when the event does not occur before B or C reacts is called non-occ and is as follows:

non-occ

$$\frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \texttt{ until } C \texttt{ then } D \xrightarrow[T_0 \setminus M]{b} B' \texttt{ until } C' \texttt{ then } D} \ (M \not\supseteq \uparrow T) \ .$$

An example of applying non-occ requires B or C to be a reactive behaviour, and one that reacts before C becomes true; say B is the behaviour in (ex-occ),

$$B \equiv \texttt{1 until (time >= 1) then 2}$$

and we denote the transition tree for B by X:

$$X \equiv \quad \frac{1 \xrightarrow[\mathbb{T} \setminus \emptyset]{t \mapsto 1} \varepsilon \quad time >= 1 \xrightarrow[\mathbb{T} \setminus \emptyset]{t \mapsto t \geq 1} \varepsilon}{\texttt{1 until (time >= 1) then 2} \xrightarrow[\mathbb{T} \setminus [1,\infty)]{t \mapsto 1} 2}$$

Using this as the before-behaviour in a reactive behaviour yields:

$$\frac{X \qquad \dfrac{\texttt{time >= 2} \xrightarrow[\mathbb{T} \setminus \emptyset]{t \mapsto t \geq 2} \varepsilon}{\texttt{time >= 2} \xrightarrow[\mathbb{T} \setminus [1,\infty)]{t \mapsto t \geq 2} \texttt{time >= 2}} \langle\text{no-change}\rangle}{B \texttt{ until (time >= 2) then 0} \xrightarrow[\mathbb{T} \setminus [1,\infty)]{t \mapsto 1} \texttt{2 until (time >= 2) then 0}}$$

In the preceding discussion we assumed that the condition does not yield bottom at any time. As discussed in Section 5.2, we can only determine when the event occurs if the condition becomes true before any times map to bottom. We can capture this by defining the set of times when the condition is bottom,

$$Bad = \{t \in T_0 \mid c(t) = \bot_{\mathbb{B}}\},$$

as in Section 5.2, and then restricting the occ and non-occ rules by adding side conditions which ensure that no bottoms have thus far been found. So, for the occ rule we must add the extra side condition

$$\uparrow T \not\supseteq \uparrow Bad,$$

and for the non-occ rule we require

$$M \not\supseteq \uparrow Bad.$$

(The set T is given by

$$T = \{t \in T_0 \mid c(t) = true\}.)$$

To deal with reactive behaviours where the condition becomes bottom before it is true we require the rule

$$\text{bad-cond} \qquad \frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \ \texttt{until} \ C \ \texttt{then} \ D \xrightarrow[T_0 \setminus \emptyset]{b'} \varepsilon}$$

where b' is the function that is like b until the condition becomes bottom, and is then bottom:

$$b' = t \mapsto \left\{ \begin{array}{l|l} b(t) & t \notin \uparrow Bad \\ \bot & \end{array} \right. .$$

The side condition for this rule just requires the condition to yield bottom at or before B or C reacts and at or before the condition is bottom;

$$\uparrow Bad \supseteq M \wedge \uparrow Bad \supseteq \uparrow T \iff \uparrow Bad \supseteq M \cup \uparrow T.$$

Notice that the side conditions for occ, non-occ and bad-cond are mutually exclusive and exhaustive. This is necessary for the rules to be deterministic because the premises are the same in all these rules.

We now have transition rules for `lift0`, `$*`, `time`, and `until-then`. The next step is to define a rule for `letbeh`.

6.8 Transitions for recursive reactive definitions

Recall that our motivation for developing the transition rules was to capture a procedure that enables us to give meanings to recursive reactive definitions such as

$$\texttt{letbeh a = 1 until time >= a then 2 in a}. \qquad (6.8)$$

A transition rule for definitions will require an environment to map variables to values. In keeping with our inference style rules we will adopt the syntax often used for deductions; if E can make a transition to E' in the environment u, we write

$$u \vdash \texttt{E} \xrightarrow[T_0 \setminus T_e]{e} \texttt{E'}.$$

The `letbeh` rule interprets a definition

```
letbeh a = E in F
```

using the transition E makes in the environment where a maps to the non-reactive evaluation of E (i.e., the value e on the top of the arrow). This gives the rule

$$\frac{[u|a:e] \vdash \texttt{E} \xrightarrow[T_0 \setminus M]{e} \texttt{E'} \quad [u|a:e] \vdash \texttt{F} \xrightarrow[T_0 \setminus M]{f} \texttt{F'}}{u \vdash \texttt{letbeh}\ a\ \texttt{=}\ E\ \texttt{in}\ F \xrightarrow[T_0 \setminus M]{f} \texttt{letbeh}\ a\ \texttt{=}\ E'\ \texttt{in}\ F'}$$

(Again, M is used in the same way as in the lift2 (+) rule.) This approach does not have the problem encountered in Section 6.3 because E is evaluated non-reactively.

It is instructive to apply these rules to Example 6.8 above. This requires a rule for lift2 (=>) analogous to the lift2 (+) rule.

6.9 Transitions for integral behaviours

In an integral behaviour, the integrand may be reactive. This presents no real difficulty, however, because we can integrate over non-reactive intervals—which are obtained using the transition rules—and then add up the pieces.

For example, using the transition rules we have already seen that

```
1 until (time >= 1) then 2
```

represents the behaviour that is $t \mapsto 1$ over the interval $\mathbb{T} \setminus [1, \infty) = [0, 1)$ and $t \mapsto 2$ afterwards. The integral of this behaviour is the integral of these two parts, adding the overall interval of $t \mapsto 1$ over $[0, 1)$ (which equals 1) to the second part:

$$t \mapsto \left\{ \begin{array}{l} \int_0^t 1.ds \\ 1 + \int_1^t 2.ds \end{array} \right| \begin{array}{l} t \in [0,1) \\ \end{array} \quad = \quad t \mapsto \left\{ \begin{array}{l} t \\ 1 + 2t \end{array} \right| \begin{array}{l} t \in [0,1) \\ \end{array}$$

Mathematically this is just integrating a discontinuous function by adding together the integrals over the continuous parts. Hence the transition rule for `integral` evaluates the integrand over a non-reactive interval and then computes the integral, that is,

$$\text{integral} \qquad \frac{A \xrightarrow[T_0 \setminus M]{a} A'}{\texttt{integral } A \xrightarrow[T_0 \setminus M]{t \mapsto \int_{\inf(T_0)}^t a(s).ds} X}$$

To complete the rule we need to define X, the behaviour for the next transition. The next transition will evaluate the behaviour from the times in M, and since integrals are cumulative we must add the integral so far (i.e., accumulated over the previous intervals) to this. So we must add

$$k = \int_{\inf(T_0)}^{\inf(M)} a(s).ds$$

to the integral of the new integrand, A'. The value k is a real number, however, and we require the representation of this number in CONTROL. The function $Real \in \mathbb{R} \to \texttt{Real}$ serves this purpose; it is the inverse of $[\![\_]\!]$ for terms of type `Real`. Using *Real* we define

$$X \equiv Real \left(\int_{\inf(T_0)}^{\inf(M)} a(s).ds \right) \texttt{ + integral A'}.$$

A similar problem arises for integrals with badly behaved integrands as for reactive behaviours with bad conditions. The integral rule above requires the side condition that a is integrable on the interval $T_0 \setminus M$. If this is not the case, then the following rule applies:

$$\text{bad-integral} \qquad \frac{A \xrightarrow[T_0 \setminus M]{a} A'}{\texttt{integral}\ \ A \xrightarrow[T_0 \setminus \emptyset]{t \mapsto \perp_{\mathbb{R}}} \varepsilon}$$

This asserts that if a is not integrable on the interval $T_0 \setminus M$, then the value of the integral expression is bottom for all times in the future. The reason for this is that integrals are cumulative and so if we do not know its value over some interval, then we cannot determine its value at any time after that interval. As an example, we cannot integrate the behaviour

```
1 / (time - 1)
```

because there is a division by 0 at time 1. Therefore the bad-integral rule applies and yields $t \mapsto \perp_{\mathbb{R}}$ for this behaviour.

6.10 Transitions for recursive integral definitions

Unlike recursive reactive definitions, recursive integral definitions do mean the solutions to the corresponding equations. For example, the definition

```
a = 1 + integral a
```

means a solution to the integral equation

$$x(t) = 1 + \int_0^t x(s).ds.$$

This equation has a unique solution, $x(t) = e^t$. In general, however, there may not be a unique solution, and such definitions denote $t \mapsto \perp_{\mathbb{R}}$.

We need to obtain integral equations from side conditions in the transition system. This means introducing mathematical variables, such as x above, and equating them to the result of the right hand side of a recursive definition in the letbeh rule. More explicitly, the right hand side of a recursive integral definition defining a variable a is evaluated in the environment where a maps to x, and the result of this is equated with x. Thus, assuming x is a new variable, the letbeh rule is modified as follows:

$$\frac{[u|a:x] \vdash \texttt{E} \xrightarrow[T_0 \setminus M]{e} \texttt{E'} \quad [u|a:x] \vdash \texttt{F} \xrightarrow[T_0 \setminus M]{f} \texttt{F'}}{u \vdash \texttt{letbeh } a \texttt{ = } E \texttt{ in } F \xrightarrow[T_0 \setminus M]{f} \texttt{letbeh } a \texttt{ = } E' \texttt{ in } F'} \; (x = e)$$

It is easy to verify that this new rule, together with the integral rule, gives the correct integral equation for the example from the start of this section. Note that for non-integrals, the introduction of the variable x is superfluous, and eliminating it yields the same result as for the previous `letbeh` rule. Therefore our new letbeh rule works as before for recursive reactive definitions.

The values above the arrows are no longer the denotations of behaviours over non-reactive intervals because they may contain free variables. These free variables are constrained by side conditions, and have a fixed value whenever the program is meaningful. So the transition rules are used to form equations, and the solutions to these equations are the denotations. This interpretation of the transition rules is explained further in Section 9.1.

Before we introduced the above rules to accommodate integrals, we still needed to solve equations to find the denotations of programs; in particular, to decide whether the occ or non-occ rule should be used it was necessary to solve a side condition. So this is a fundamental feature of our semantics, and not due solely to integration. Of course, solving equations—particularly integral equations—may involve very difficult mathematical analysis. This was observed in Section 5.6 in the context of plain integrals, and clearly allowing integral equations greatly increases the difficulty of the analysis.

6.11 Avenue on delayed switching

In Section 5.8 we considered alternative semantics for `until-then` which were still exact in the sense that there was not a finite length of time between the times when the condition becomes true and the reactive behaviour switching. It is now worth reconsidering these alternatives in relation to recursive reactive definitions in case they provide a simpler semantics.

In Section 6.3 we proved that some recursive reactive programs have no meaning if we take the view that recursive definitions are equations. This is the usual interpretation of recursive definitions, so it is perhaps more satisfactory to change the semantics of `until-then` than devise a new interpretation of recursive definitions. However, we shall see that none of the alternative semantics for `until-then` make this possible.

There are two possible alternative semantics for `until-then` which we called Fran and Early. We will consider only the Fran alternative in this section; the analysis for Early is similar.

The Fran alternative uses Elliott and Hudak's semantics for `untilB` and `predicate` in Fran for `until-then` in CONTROL,

$$[\![B \texttt{ until } C \texttt{ then } D]\!] t_0\, u \;=\; t \mapsto \left\{ \begin{array}{l|l} [\![B]\!] t_0\, u\, t & t \leq t_e \\ [\![D]\!] t_e\, u\, t & \end{array} \right.$$

$$\begin{array}{rrcl} \text{where} & T & = & \{t \in \mathbb{T} \mid t \geq t_0 \wedge [\![C]\!] t_0\, u\, t\} \\ & t_e & = & inf(T) \end{array}$$

Here the values t_0 and t_e are times; these replace the sets of times used in our semantics because every event has an event time. Note that we have ignored the possibility that the condition may yield $\perp$ at some times. With this semantics a reactive behaviour switches from B to D strictly after the infimum of the times when C is true (t_e). Part of the reason for switching strictly after is to avoid the problem with recursive definitions. The idea is that a recursive reactive behaviour of the form

$$\texttt{letbeh}\ a\ \texttt{=}\ B\ \texttt{until}\ C[a]\ \texttt{then}\ D\ \texttt{in}\ F$$

always yields B at the time when the event occurs, and so the condition (in terms of a) is not affected by a switching. For example, given the term

```
letbeh b = 1 until (time >= b) then 2 in b
```

we obtain the corresponding equations:

$$\begin{aligned} x &= t \mapsto \left\{ \begin{array}{c|l} 1 & t \leq t_e \\ 2 & \end{array} \right. \\ t_e &= \inf(\{t \in \mathbb{T} \mid t \geq x(t)\}) \end{aligned}$$

This has one solution,

$$x = t \mapsto \left\{ \begin{array}{c|l} 1 & t \leq 1 \\ 2 & \end{array} \right.$$

This approach looks very promising until we consider some other examples. The following program does not have any meaning under this semantics:

$$\texttt{letbeh b = 1 until (time > b) then 2 in b} \tag{6.9}$$

In other words, there are no solutions to the following equations:

$$\begin{aligned} x &= t \mapsto \left\{ \begin{array}{c|l} 1 & t \leq t_e \\ 2 & \end{array} \right. \\ t_e &= \inf(\{t \in \mathbb{T} \mid t > x(t)\}) \end{aligned}$$

One could be argued that such programs should not have any meaning, but there is a more serious problem. Consider the program

$$\texttt{letbeh b = 1 until (time > b) then 0 in b} \tag{6.10}$$

which gives the equations,

$$
\begin{aligned}
x &= t \mapsto \left\{ \begin{array}{c|l} 1 & t \leq t_e \\ 0 & \end{array} \right. \\
t_e &= \inf(\{t \in \mathbb{T} \mid t > x(t)\})
\end{aligned}
$$

These equations have a solution, which is anomalous because 6.9 is the same program as 6.10 except for the after-behaviour. So under this semantics the after-behaviour can influence whether the event can be determined or not. This breaks the causality principle that things in the future cannot affect the present.

In summary, Elliott and Hudak's semantics for `untilB` and `predicate` in Fran can be used for `until-then` in CONTROL, but this does not provide a reasonable semantics for recursive definitions under the equational interpretation of definitions. In fact, it is worse than using our semantics for `until-then` because although it gives a meaning to more programs it violates the causality principle.

Chapter summary

Some behaviours can only be expressed if we can refer to them in their own definition, in other words, if we can define them recursively. In particular, in some reactive behaviours the condition needs to refer to the behaviour itself.

This technique is not an equational approach to defining behaviours recursively because there are no solutions to the resulting equations. The intended meaning of such definitions relies on the operational notion of non-reactive evaluation, and this can be formalised by a transition system. Integrals can also be accommodated by the transition system so that recursive integral definitions make sense.

Chapter 7

Functions and behaviours

So far we have seen operators for constructing behaviours, but we have no facilities for:

- Parameterising a behaviour by a variable.
- Constructing a periodic, or repeating, behaviour (or more generally a behaviour with an infinite number of states).

Most programming languages provide parameterised expressions that can be named and referred to elsewhere. This allows the same expression to be re-used with different values of the parameter. These parameterised expressions are often called functions because they behave similarly to functions in mathematics. Another ingredient universal in programming is repetition, often provided for by recursion in functional languages and by loops in imperative languages. Recursion in CONTROL makes it possible to describe periodic behaviours; that is, behaviours that repeat the same values over some interval, as `sin` does. In imperative and functional programming the combination of parameterising code by variables and repetition is essential for many programs. We will see that this also applies to CONTROL when we introduce functions and recursion.

Functions and recursion are key aspects of PCF, and the mechanisms we adopt for CONTROL are the same. However, in addition to recursive functions CONTROL has recursive behaviours which we discussed in Chapter 6. These two mechanisms for recursive definitions are quite different, but, as we shall see, they complement each other to provide a powerful programming paradigm. In this chapter we show how the semantics for the separate mechanisms can be unified within one language.

7.1 Functions

Consider the term:

```
integral (time * lift0 2) + integral (time * lift0 3).
```
(7.1)

Assuming that we can define a function `f` of a variable `x` by

```
f x = integral (time * lift0 x),
```
(7.2)

then Term 7.1 can be re-expressed as

```
f 2 + f 3.
```

This saves writing almost the same expression twice.

We will use λ-notation for functions in CONTROL. A term

$$\lambda x.L$$

represents a function that takes an argument and yields the term L with the value of the argument substituted for all free occurrences of x. With this notation we would write `f` from (7.2) as follows:

```
f = λx.  integral (time * lift0 x).
```

In some languages there is a `let` construct so that functions can be defined and then used within the main program term. Using such a facility Term 7.1 could be written as

```
let  f = λx.  integral (time * lift0 x)
in   f 2 + f 3.
```

In general, the `let` construct binds a variable, f, to a term, F, within a body, M,

$$\texttt{let}\ f\ \texttt{=}\ F\ \texttt{in}\ M.$$

However, we already have a mechanism for binding variables within a term, λ-abstraction. The `let` notation is equivalent to an abstraction, to abstract over f in M, and an application, to apply F to this abstraction. Lambda notation therefore subsumes `let`, and we can define `let` as syntactic sugar as follows:

$$\texttt{let}\ f\ \texttt{=}\ F\ \texttt{in}\ M \quad\equiv\quad (\lambda f.M)F.$$

We can also define the function definition notation we used in (7.2) as syntactic sugar:

$$\texttt{let}\ f\ n\ \texttt{=}\ F\ \texttt{in}\ M \quad\equiv\quad \texttt{let}\ f\ \texttt{=}\ \lambda n.F\ \texttt{in}\ M.$$

We will now give a brief description of the syntax and semantics of λ-terms. Firstly, recall the syntax of variables, λ-abstractions and applications from Chapter 4,

Types	θ	$::=$	θ `->` θ	function types
Terms	E	$::=$	x	variables
		$\mid$	$\lambda x{:}\theta.E$	λ-abstractions
		$\mid$	$E\ E$	applications.

Notice that a type must be supplied for λ-bound variables. In the above discussion we omitted types for simplicity, but from now on types will be given for all bound variables.

Both the operational and denotational semantics of λ-terms are important in our theory of CONTROL. The following informal descriptions are made precise in Chapter 8.

The operational semantics of λ-terms is based on β-reduction, which defines the application of a function (i.e., an abstraction) to an argument to be the result of substituting the argument for the bound variable,

$$\beta - \text{rule} \qquad\qquad (\lambda x : \theta.L)N \to L/[x : N],$$

where $L/[x : N]$ denotes substituting the term N for free occurrences of the variable x in L.

The denotational semantics requires an environment to be passed to the semantic function. The environment assigns a value to every free variable in the term. The semantic equations are as follows, where x is a variable and u is an environment: for variables,

$$[\![x]\!]u = ux;$$

λ-abstractions represent actual functions,

$$[\![\lambda x : \theta.L]\!]u = d \mapsto [\![L]\!][u|x : d];$$

and applications are defined as function application,

$$[\![MN]\!]u = ([\![M]\!]u)([\![N]\!]u).$$

(The formulas on the right hand side are guaranteed to be type correct by our use of the simple type system.)

7.2 Recursive functions

Although λ-abstractions are convenient, they do not by themselves add much expressiveness to the language because of the restrictions imposed by the simple type system. In particular, it is not possible to describe repeated patterns of computation, such as a behaviour that repeats over some interval. In this section we will explain this point and introduce a recursion operator that allows a function to call itself, which enables repeated computation. Operationally this operator is straightforward, but denotationally it is more difficult to capture. However, we will show that the usual treatment of recursive functions is compatible with our semantics of behaviour operators (this does not include recursive behaviours as discussed in Chapter 6).

A recursive definition has the form

$$f \;\texttt{=}\; F \tag{7.3}$$

where the function f recurs on the right hand side, that is, in F. If we wrote this definition using `let` then the occurrence of f on the right hand side would be free, and not associated with the function f being defined.

In the untyped λ-calculus it is possible to define recursion combinators such as

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

They have the property that

$$YG \to G(YG), \tag{7.4}$$

and this allows recursive definitions (7.3) to be written as

$$Y(\lambda f.F).$$

To see how this definition provides recursion, consider the following reduction:

$$
\begin{array}{ll}
 & Y(\lambda f.F) \\
\rightarrow & \langle \text{by } 7.4 \rangle \\
 & (\lambda f.F)(Y(\lambda f.F)) \\
\rightarrow & \langle \beta\text{-reduction} \rangle \\
 & F/[f : (Y(\lambda f.F))].
\end{array}
$$

In the function body F the variable f is replaced by $Y(\lambda f.F)$ which is the definition of f, and by the same reduction sequence the definition of f can be unwound as many times as necessary. The combinator Y is not a valid term of the simply typed λ-calculus because xx is untypable (x is a function and its own argument). In fact, all recursion combinators are untypable in the simply typed λ-calculus, and therefore we need a built in recursion operator in CONTROL to enable us to write recursive functions.

We could define a recursion operator, say `rec`, that allows us to write recursive function in the same way that Y does in the untyped λ-calculus,

$$\texttt{rec }(\lambda f : \theta.F).$$

But in such definitions we always write an abstraction to abstract over f in F. Therefore an alternative is to define a binding construct, μ, which binds a variable within a function body recursively, giving the equivalent term

$$\mu f : \theta.F$$

We prefer this construct because it avoids an extra λ-abstraction and makes it clear which variable is bound recursively.

As for λ-abstractions, we can define a let mechanism as syntactic sugar:

$$\begin{aligned}\texttt{letrec}\ f\ \texttt{=}\ F\ \texttt{in}\ M &\equiv \texttt{let}\ f\ \texttt{=}\ \mu f:\theta.F\ \texttt{in}\ M\\ &\equiv (\lambda f:\theta.M)(\mu f:\theta.F).\end{aligned}$$

Notice that we require a type for f on the right hand side. Strictly speaking, types should be given in all `letrec` definitions, but as `letrec` is not part of the formal language we will not do so.

Operationally the semantics of recursive definitions is very simple; we require the following reduction rule to unwind a recursive function one level:

$$\mu\text{-rule} \qquad \mu f:\theta.F \ \rightarrow\ F/[f:(\mu f:\theta.F)]$$

This rule gives the equivalent term in the simply typed λ-calculus to reduction on $Y(\lambda f.F)$ in the untyped λ-calculus.

Denotationally the semantics of μ is more complicated. We discussed domains for function types in Chapter 4 and described a domain structure that ensures recursive definitions have a unique meaning. We will complete the picture by showing how that theory enables us to define the meaning of recursive definitions.

In functional languages a recursive function definition such as

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

is interpreted as the solution to an equation involving an unknown f

$$f(n) \ = \ \left\{ \begin{array}{l} 1 \\ n \times (f(n-1)) \end{array} \right| \begin{array}{l} n = 0 \\ \ \end{array}$$

This is why functional languages are declarative—definitions are equations that always hold, so the right hand side can always be substituted for the function. So the meaning of `fact` is a solution for f in this equation, but unfortunately there are many solutions:

$$f_\perp(n) \quad = \quad \left\{ \begin{array}{l|l} n! & n \geq 0 \\ \perp & \end{array} \right.$$

$$f_0(n) \quad = \quad \left\{ \begin{array}{l|l} n! & n \geq 0 \\ 0 & \end{array} \right.$$

$$f_x(n) \quad = \quad \left\{ \begin{array}{l|l} n! & n \geq 0 \\ x & n = -1 \\ (-1)^{n-1} x \times \prod_{i=1}^{|n|-1} \frac{1}{i} & \end{array} \right.$$

The last solution f_x is valid for any value of x, so there are infinitely many solutions.

In general there are many solutions to equations arising from recursive definitions, but there is only ever one solution that agrees with the operational interpretation of recursive definitions. Operationally, we evaluate a recursive definition by unwinding the function body and substituting for the argument each time a recursive call is made. The solution that agrees with evaluation is always the one that satisfies the equations and terminates for as few arguments as possible, that is, the least defined solution (for the example above this is $f_\perp$). Intuitively this is because the term does not contain any information about the result when computation loops indefinitely, so the result should be bottom—the least defined value. For example, there is nothing in the definition of `fact` to suggest that the result should be 0, or any other number, for negative arguments.

Suppose that g_0 is the meaning of the recursive definition $\mu f : \theta.F$ in some environment u; that is,

$$g_0 \quad = \quad [\![\mu f : \theta.F]\!]u \tag{7.5}$$

The meaning should remain the same after a μ-reduction step; that is, the denotational semantics should agree with the operational semantics. Thus,

$$
\begin{aligned}
& g_0 \\
= \; & \langle \text{by } 7.5 \rangle \\
& [\![\mu f : \theta . F]\!] u \\
= \; & \langle \mu \rangle \\
& [\![F/[f : (\mu f : \theta . F)]]\!] u \\
= \; & \langle \text{substitution} \rangle \\
& [\![F]\!] [u | f : [\![(\mu f : \theta . F)]\!]] \\
= \; & \langle \text{by } 7.5 \rangle \\
& [\![F]\!] [u | f : g_0].
\end{aligned}
$$

Therefore g_0 is certainly a solution to the following equation in g:

$$g \;\; = \;\; [\![F]\!] [u | f : g].$$

This equation is usually expressed equivalently as finding a fixed point of the function

$$G = g \mapsto [\![F]\!] [u | f : g].$$

The solution we require, g_0, is certainly a fixed point of G, but in general there may be many fixed points. Fortunately, there is a way of selecting the one that corresponds to our operational semantics—the solution we require is always the least fixed point with respect to the ordering on domains we defined in Section 4.2. Moreover, the following theorem guarantees that for any G arising from a recursive definition, there must be a least fixed point.

THEOREM 7.1 (LEAST FIXED POINT) *If $G : D \to D$ is an ω-continuous function on the CPO D, then G has a least fixed point given by*

$$g_0 \;\; = \;\; \bigsqcup_{n=0}^{\infty} G^n(\bot) \qquad \Box$$

So the Least Fixed Point Theorem guarantees that all recursive definitions can be assigned a meaning, and provides a formula for these meanings. Because of the ordering on domains, we expect these meanings to correspond to the functions obtained from an operational perspective, and it can be shown that this is so.

7.3 Examples of recursive functions

Using recursive functions we can write many new behaviours that cannot be expressed without them. For example, the following recursive function defines a reactive behaviour that increments by one as each second passes:

```
letrec a n = n until (time>=n+1) then a(n+1) in a 0
```

The right hand side of the definition is a reactive behaviour that yields n until the time equals $n+1$. At this time it calls itself with the argument $n+1$, so every second the behaviour increases by 1. If λ-abstractions or recursive functions were not part of CONTROL, and no other mechanisms were introduced, it would not be possible to write a program which yields an equivalent behaviour.

Formally we can interpret the above program operationally and denotationally given the semantic techniques introduced thus far. We begin by desugaring the program as follows (we omit type declarations on λ and μ

bound variables for clarity):

$$
\begin{array}{ll}
 & \texttt{letrec a n = n until (time>=n+1) then a(n+1) in a 0} \\
\equiv & \\
 & \texttt{letrec a = } \underbrace{\lambda\texttt{n.n until (time>=n+1) then a(n+1)}}_{A} \texttt{ in a 0} \\
\equiv & \\
 & \texttt{let a = } \mu\texttt{a.}A \texttt{ in a 0} \\
\equiv & \\
 & (\lambda\texttt{a.a 0})\ (\mu\texttt{a.}A)
\end{array}
$$

The final line is a program in the core syntax, and therefore it can be interpreted operationally by the β and μ rules:

$$
\begin{array}{ll}
 & (\lambda\texttt{a.a 0})\ (\mu\texttt{a.}A) \\
\rightarrow & \langle\beta\rangle \\
 & (\mu\texttt{a.}A)\texttt{ 0} \\
\rightarrow & \langle\mu\rangle \\
 & (A/[\texttt{a:}\mu\texttt{a.}A])\texttt{ 0} \\
\equiv & \\
 & (\lambda\texttt{n.n until (time>=n+1) then }(\mu\texttt{a.}A)\texttt{(n+1))0} \\
\rightarrow & \langle\beta\rangle \\
 & \texttt{0 until (time>=0+1) then }(\mu\texttt{a.}A)\texttt{ (0+1)} \\
\equiv & \\
 & \texttt{0 until (time>=1) then }(\mu\texttt{a.}A)\texttt{ 1}
\end{array}
\tag{7.6}
$$

We now have an `until-then` term at the top-level. This can be evaluated using the transition rules from Chapter 6, and we find that the behaviour yields 0 over the interval [0, 1) and then behaves like $(\mu\mathtt{a}.A)\ \mathtt{1}$. The term $(\mu\mathtt{a}.A)\ \mathtt{1}$ is exactly the same as Term 7.6 above except 1 replaces 0. Thus, by replacing 0 with 1 in the above evaluation it is straightforward to calculate that over the interval [1, 2) the behaviour yields 1 and then behaves like $(\mu\mathtt{a}.A)\ \mathtt{2}$. An inductive argument can be used to find the complete meaning of the program, effectively capturing the repetition of this procedure.

We will now interpret the same program denotationally, beginning with the unsugared program,

$$(\lambda\mathtt{a}.\mathtt{a}\ \mathtt{0})\ (\mu\mathtt{a}.A) \tag{7.7}$$

We need to apply the semantic equations for λ and μ, but our semantic function also requires the set of times when the behaviour is alive, initially $\mathbb{T}$, as discussed in Chapter 5. Thus,

$$\begin{aligned}
& [\![(\lambda\mathtt{a}.\mathtt{a}\ \mathtt{0})\ (\mu\mathtt{a}.A)]\!](\mathbb{T})[] \\
= \quad & \langle \lambda \text{ and } \mu \rangle \\
& (d \mapsto [\![\mathtt{a}\ \mathtt{0}]\!](\mathbb{T})[a : d])(\bigsqcup_{n=0}^{\infty}(g \mapsto [\![A]\!](\mathbb{T})[a : g])^n \bot) \\
= \quad & \\
& (d \mapsto d(0))(\bigsqcup_{n=0}^{\infty}(g \mapsto [\![A]\!](\mathbb{T})[a : g])^n \bot) \\
= \quad & \\
& (\bigsqcup_{n=0}^{\infty}(g \mapsto [\![A]\!](\mathbb{T})[a : g])^n \bot)0
\end{aligned} \tag{7.8}$$

Next we must evaluate $g \mapsto [\![A]\!](\mathbb{T})[a:g]$. We will do this as a side step:

$$
\begin{array}{ll}
& g \mapsto [\![A]\!](\mathbb{T})[a:g] \\
= & \\
& g \mapsto [\![\lambda\texttt{n.n until time>=n+1 then a(n+1)}]\!](\mathbb{T})[a:g] \\
= & \langle\lambda\rangle \\
& g \mapsto d \mapsto [\![\texttt{n until time>=n+1 then a(n+1)}]\!](\mathbb{T})[a:g|n:d] \\
= & \langle\text{until-then}\rangle \\
& g \mapsto d \mapsto t \mapsto \left\{ \begin{array}{l} [\![\texttt{n}]\!](\mathbb{T})[a:g|n:d] \\ [\![\texttt{a(n+1)}]\!](\uparrow T)[a:g|n:d] \end{array} \right| [\![\texttt{time>=n+1}]\!](\mathbb{T})[a:g|n:d] \\
= & \\
& g \mapsto d \mapsto t \mapsto \left\{ \begin{array}{l} d \\ g(d+1) \end{array} \right| t \geq d+1
\end{array}
$$

Now substituting this formula in the Equation 7.8 yields the value of Term 7.7 as follows:

$$
\begin{array}{ll}
& [\![\texttt{(}\lambda\texttt{a.a 0) (}\mu\texttt{a.}A\texttt{)}]\!](\mathbb{T})[] \\
= & \\
& (\bigsqcup_{n=0}^{\infty}(g \mapsto d \mapsto t \mapsto \left\{ \begin{array}{l} d \\ g(d+1) \end{array} \right| t \geq d+1 \quad)^n \bot)0
\end{array}
$$

7.4 Recursive behaviours revisited

In Chapter 6 we studied a construct for recursive behaviour definitions with the syntax

```
letbeh a = B in F.
```

This is similar to the syntax for `letrec` and we can define `letbeh` as syntactic sugar in terms of a recursive binding construct, ν, and a λ-abstraction (as we did for `letrec`) as follows:

$$\texttt{letbeh a = B in F} \equiv \texttt{let a = } \nu a : \theta.B \texttt{ in F}$$
$$\equiv (\lambda a : \theta.F)(\nu a : \theta.B)$$

The example from Section 6.3,

`letbeh a = 1 until (time >= a) then 2 in a`,

can then be expressed as

ν`a:Beh Real.1 until (time >= a) then 2`.

To interpret this behaviour we must use the transition rules because they capture non-reactive evaluation and this is essential for making sense of terms such as this one. In other words, we can find the meaning of such terms using the operational semantics, but not using the purely denotational methods. The denotational semantics developed in Chapter 5 and in this chapter does not account for the complete language because it does not interpret ν definitions. For this reason our complete semantics in the next chapter will only describe the operational style semantics based on the transition rules. It is unlikely that any compositional denotational semantics extending ours could capture ν-definitions because of the problem discussed in Chapter 6 with recursive reactive behaviours.

7.5 Combining recursive behaviours and recursive functions

We now have described two mechanisms for defining a behaviour recursively: `letbeh` and `letrec`. They both work very differently, and have different semantics. In this section we will discuss the following points regarding these recursion mechanisms:

- Why we need both.
- When each one should be used.
- How both can be used in the same program.
- The combined semantics.

We will consider examples to demonstrate why both mechanisms are needed. There is no way to program

```
letbeh a = 1 until (time >= a) then 2 in a
```

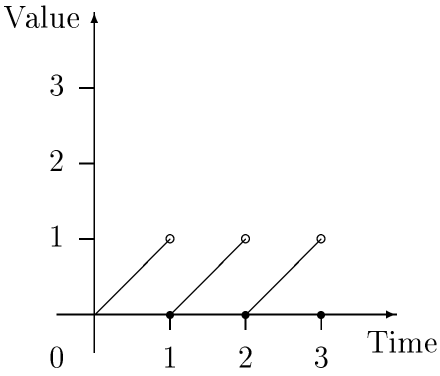
using `letrec` because there are no solutions to the corresponding equations (as we saw in Section 6.3). This was the motivation for the `letbeh` construct. It is essential that `a` on the right hand side of the definition refers to behaviour being defined, and unwinding the definition by replacing `a` by the right hand side, as `letrec` does, leads to an infinite regression. This program is not describing a repeating pattern, as `letrec` does, but rather it assumes the existence of a behaviour object, `a`, and refers to this object in its own definition. As we saw in Chapter 6, the method of non-reactive evaluation allows us to interpret such definitions correctly.

Similarly, we cannot write the program

```
letrec a n = n until (time>=n+1) then a(n+1) in a 0
```

using `letbeh`. The `letbeh` construct is only defined for behaviours, and in this definition `a` is a function from numbers to behaviours. Furthermore, there is no simple extension of `letbeh` to account for functions because this interpretation would suggest there are an infinite number of behaviour objects, one for each value of the function argument. But recursive definitions such as this example do not refer to the behaviour with the same value of the argument, so no behaviour object refers to itself and the `letbeh` mechanism is therefore unnecessary. The usual semantics of `letrec` gives exactly the interpretation we have in mind for such functions.

We will now consider when we should use each mechanism. When we have an actual behaviour object that can only be defined in terms of itself then we must use `letbeh`. When we have a function that yields a behaviour when supplied with an argument, and that behaviour is reactive and calls itself in when the behaviour reacts, we must use `letrec`. In fact, we do not necessarily require a function to write a `letrec` definition. Consider the behaviour that yields the time until the time is 1, and then repeats these values every second. Its graph is a saw wave that increases linearly with gradient 1 for one second, and then drops instantly back to 0:



This can be described by the following term:

```
letrec a = (integral 1) until (integral 1 >= 1) then a in a.
```

We use `letrec` here because we are describing a repeating behaviour. This is similar to the program ones,

```
letrec ones = 1 :  ones in ones,
```

in a lazy functional language with lists (e.g., Haskell), which yields the infinite list of ones. Both programs define a recursive value rather than a recursive function.

It is useful to use both recursion mechanisms in the same program as the next example demonstrates:

```
letbeh
  p = integral
      (
        letrec
          v =  1 until p >= 1 then
              -1 until p <= -1 then v
        in v
      )
in p
```

This behaviour is 0 initially and increases at a rate, `v`, of 1 until it reaches 1. Then it increases at a rate of -1 until it reaches -1 and then increases at a rate of 1 again. Thus it is a triangle wave with amplitude 1 and period 4. Graphically we may think of this program as describing the horizontal position of a ball bouncing (elastically) between two walls at -1 and 1, and travelling at a constant speed of 1.

It is not difficult to write an equivalent behaviour that does not use `letbeh` because it is easy to work out what `p` is on the right hand side and substitute it for an equivalent behaviour. However, if in the above program we add some term to the integral expression then it becomes more difficult to solve for `p`, and in some cases impossible. Therefore many slight variations of the above program make essential use of both `letbeh` and `letrec`.

Our final question concerns the semantics of programs like the example above that use both recursion mechanisms in the same program. As we said earlier, our denotational semantics does not account for `letbeh` so we must use the operational semantics provided by the transition system. To accommodate λ-abstractions and recursive definitions we need to extend the transition system. The method for doing this derives from the example in Section 7.3. A behaviour term that is an application or recursive function at the top-level must be evaluated by first performing some evaluation steps using the β and μ rules. This is repeated until a behaviour is obtained at the top-level; that is, the top-level syntactic construct is `lift0`, `$*`, `until-then`, `integral` or ν. Then the transition rules can be applied to interpret the term. This can be achieved by adding the following rule to the transition system:

$$\text{reduce} \qquad \frac{E \to E'' \quad E'' \xrightarrow[T_0 \setminus M]{e} E'}{E \xrightarrow[T_0 \setminus M]{e} E'}$$

A term E that is an application or recursive function is first evaluated one step to the term E'', and then if E'' can make a tranistion to E' the overall term can make this transition. Note that the reduce rule may need to be applied repeatedly, as many times as necessary to obtain a behaviour at the top-level. The evaluation relation $\to$ is defined precisely in the next chapter.

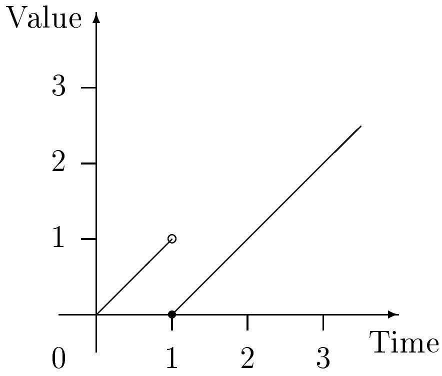
7.6 Local and global time

In this section we provide definitions of local and global time behaviours. This serves three purposes: first, it demonstrates that the behaviour `time` does not need to be a primitive; second, that we can define local time (the time since the enclosing behaviour came alive); and third, it further illustrates the difference between `letbeh` and `letrec`.

The following program defines a behaviour `ltime` that gives the time since the enclosing behaviour came alive; that is, the local time. By using this behaviour in a reactive behaviour we can observe the semantics of local time, for example,

```
letrec ltime = integral 1
in     ltime until (time >= 1) then ltime
```

By applying the semantics it is straightforward to show that the graph corresponding to this behaviour is

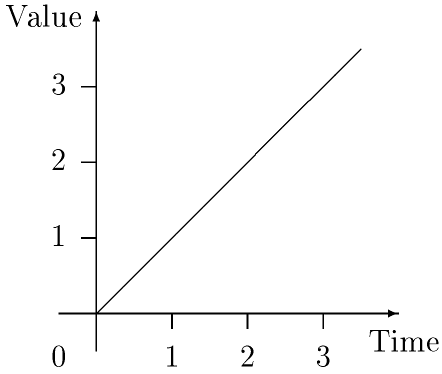


Intuitively, `letrec` creates a recursive binding for `ltime` and each occurrence in the main body is unwound by the μ rule. Therefore the second `ltime` is a new behaviour that is alive from time 1 onwards, and it equals the integral of 1 from time 1 onwards. In general, behaviours defined using `letrec` produce a family of behaviours, one for each occurrence of the behaviour in the main body, and each one comes alive when the enclosing behaviour comes alive. This explains why a behaviour such as `ltime` seems to be reset and start integrating from zero again after the event. We would expect functions that yield behaviours to work this way because when such functions are applied to an argument a new behaviour results, but the same applies to behaviour values such as in the example above.

Now let us write the same program but with `letbeh` instead of `letrec` and `gtime` instead of `ltime`,

```
letbeh gtime = integral 1
in     gtime until (time >= 1) then gtime
```

Applying the semantics yields a behaviour whose graph is:



In this program the second occurrence of the behaviour `gtime` is not reset at time 1. Intuitively, a behaviour defined using `letbeh` is an object, and that object is the same wherever it is referred to in the main body. Examining the semantics we can confirm this because the ν transition rule updates the body (in this example, `integral 1`) in the new behaviour, thus a `letbeh` definition lifts a behaviour out of the main body and ensures that all references to this behaviour yield this value.

7.7 Multiple definitions

We have seen that both `letbeh` and `letrec` are useful recursion mechanisms and that sometimes we need both in the same program. To allow recursive functions and behaviours that are mutually recursive, we require a form of definition that enables us to give both simultaneously. In PCF multiple mutually recursive definitions can be dealt with by forming a tuple of the variables and a tuple of the right hand sides, and forming a single recursive definition (see [Rey98, pp. 301] or [Mit96, pp. 64]). For example, for two mutually recursive definitions:

$$\begin{array}{l} \texttt{letrec}\ f \ \texttt{=}\ F[f,g] \\ \qquad\quad\ g \ \texttt{=}\ G[f,g] \\ \quad \texttt{in}\ M \end{array} \quad\equiv\quad \begin{array}{l} \texttt{letrec}\ (f,g)\ \texttt{=}\ (F[f,g],G[f,g]) \\ \qquad\quad \texttt{in}\ M \end{array}$$

This will not work for us because we have two different recursion mechanisms, and so multiple definitions cannot be reduced to a single definition using tuples.

There is another standard method which reduces multiple definitions into a nested definition (see [Ten91, pp. 111] or [Mit96, pp. 338]). For two mutually recursive definitions the translation is as follows:

$$\begin{array}{l} \texttt{letrec}\ f \ \texttt{=}\ F[f,g] \\ \qquad\quad\ g \ \texttt{=}\ G[f,g] \\ \quad \texttt{in}\ M \end{array} \quad\equiv\quad \begin{array}{l} \texttt{letrec}\ f\ \texttt{=}\ \ \texttt{letrec}\ g\ \texttt{=}\ G[f,g] \\ \qquad\qquad\qquad \texttt{in}\ F[f,g] \\ \texttt{in}\ \ \texttt{letrec}\ g\ \texttt{=}\ G[f,g] \\ \qquad \texttt{in}\ M \end{array}$$

The right hand side of the definition for f defines g so that it can be referred to in the expression $F[f,g]$. The main body also defines g, this time so that it can be referred to in the term M. The variable f can be referred to on the right hand side of its definition, because `letrec` allows this, and in the main body of course. Therefore this translation preserves the meaning of the overall term. The only slight drawback is that the definition of g is duplicated.

This translation works for multiple `letrec` and `letbeh` definitions because the appropriate construct can be used for each one. Thus, we can add a general `let` mechanism for multiple definitions, which allows either recursion mechanism to be used for each definition by tagging the definition; for example:

```
let beh a = A
    rec f = F
in M.
```

This program defines a recursive behaviour a, and a recursive function f, and they can be mutually recursive. Such programs can be translated into the core syntax using the method described above. In this case eliminating the multiple definitions gives

```
letbeh a = letrec f = F
           in A
in letrec f = F
   in M.
```

The translation may be continued by desugaring `letbeh` and `letrec` to obtain a term in the core syntax.

7.8 Avenue on Zeno

Many reactive systems run forever, sending outputs continually and reacting to inputs. This is different to non-termination when a program gets into an infinite loop, because outputs are always being produced. An example of such a program is the one we saw earlier which increments a behaviour by one each second. It is also possible to write programs that become stuck temporally, yet never stop producing output. Here is an example:

```
letrec zeno n = n until time>=((n-1)/n) then zeno(n+1)
in zeno 1
```

In this program a sequence of events occur at times $\frac{0}{1}, \frac{1}{2}, \frac{3}{4}, \frac{4}{5}, \ldots$, and each time the value of the behaviour increases by 1. The limit of this sequence of event times is 1, so there are infinitely many events that occur before time 1. We cannot say anything about the value of this behaviour at or after time 1 because it never reaches time 1.

Semantically we should regard values of this behaviour after time 1 to be $\perp$. However, our semantics cannot make this explicit because it is not possible in general to determine when a behaviour will become stuck temporally. We can use our semantics to reason about the value of terms such as these, but unfortunately it is not true in general that our semantics gives the value of behaviours at all times. Although this is a limitation, it is no worse than being unable to determine which values terminate in PCF. Furthermore, in many cases, such as for `zeno`, using meta-level reasoning we are able to identify when a behaviour will become stuck temporally.

Chapter summary

The functional subset of CONTROL has the same operational and denotational semantics as PCF. In this chapter we saw how the operational semantics can be combined with our semantics of behaviours, and in particular how the two different recursion mechanisms can be incorporated. This involved adding a rule to the transition system for reducing λ-abstractions and recursive functions. Then we considered a general let construct for multiple mutually recursive `letrec` and `letbeh` definitions. This is based on a translation that converts multiple mutually recursive definitions into a nested recursive definition. This way terms can be translated into the core syntax and evaluated by the operational semantics.

Chapter 8

Complete formal semantics

In this chapter we present a complete formal semantics for CONTROL. This brings together the development from the preceding chapters, and formalises the type system, evaluation rules and transition rules. The semantics is described bottom up so that all parts are defined before they are used. These parts are then brought together to give a semantics for terms.

8.1 Syntax

The abstract syntax of CONTROL is as follows (x represents variables):

$$
\begin{array}{lll}
\text{Types} & \theta & ::= \ \texttt{Real} \mid \texttt{Bool} \mid \theta\texttt{->}\theta \mid \texttt{Beh}\ \theta \\
\text{Constants} & K & ::= \ \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{true} \mid \texttt{false} \mid \texttt{+} \mid \ldots \mid \texttt{if}_\theta \mid \ldots \\
\text{Terms} & E & ::= \ K \mid x \mid \lambda x\!:\!\theta\,.\,E \mid E\,E \mid \mu x\!:\!\theta\,.\,E \mid \\
 & & \quad\ \texttt{lift0}\ E \mid E\ \texttt{\$*}\ E \mid \texttt{integral}\ E \mid \\
 & & \quad\ E\ \texttt{until}\ E\ \texttt{then}\ E \mid \nu x\!:\!\theta\,.\,E
\end{array}
$$

This abstract grammar is ambiguous if interpreted as a concrete context-free grammar. Therefore, when we write terms they must be fully parenthesised to avoid ambiguity. However, we can relax this requirement, and thereby make terms more readable, by declaring the precedence and associativity of each construct. (This is useful for discussing terms, but is not part of the formal language description.) The constructs listed in descending order of precedence, and grouped into constructs with equal precedence, are as follows:

Types `Beh, ->`

Terms `(lift0, integral), $*, until-then,` (λ, μ, ν)

As usual, function types associate to the right and function application associates to the left—similarly for behaviour application. The `until-then` operator is neither left nor right associative. The constants assume their usual precedence and associativity.

Built in functions are included as constants. A full list of constants is given in Appendix A.

Free variables are defined as follows:

Definition The set of free variables of a term E is given by $FV(E)$, which is defined by the following equations:

$$FV(0) = \{\}$$

(and similarly for all constants),

$$FV(x) = \{x\}$$

$$FV(\lambda x\!:\!\theta\,.\,L) = FV(L) \setminus \{x\}$$

(and similarly for μ and ν),

$$FV(MN) = FV(M) \cup FV(N)$$

(and similarly for all the remaining constructs). □

A term that has no free variables is *closed*, and otherwise it is *open*. A CONTROL *program* is a closed term.

We sometimes want to distinguish non-behaviour terms from behaviour terms. The set of non-behaviour terms can be defined syntactically, rather than via types, as follows:

$$\text{NonBeh} \quad E \quad ::= \quad K \mid x \mid \lambda x\!:\!\theta\,.\,E \mid E\,E \mid \mu x\!:\!\theta\,.\,E.$$

There are no behaviour constants in CONTROL.

8.2 Type system

A *typing judgement* asserts that a term E has type θ in a context Γ, and is written as

$$\Gamma \vdash E : \theta.$$

The context Γ gives the types of the free variables in E. Contexts are defined as follows:

Definition A context $\Gamma \in$ Context is a partial function from variables to types:

$$\text{Context} = \text{Variable} \rightharpoonup \text{Type}. \qquad \square$$

A context Γ is valid for a term E if it assigns types to all the free variables in E, that is, if

$$FV(\text{dom}\ \Gamma) \supseteq FV(E).$$

We use inference rules to specify which typing judgements are valid; that is, valid typing judgements in our type system are those that can be derived using the inference rules given in Figure 8.1. We use standard inference style rules (see [Ten91] or [Car97]) with one notational shorthand: if the type assignment is the same for all the premises as for the conclusion, then we leave it out.

All constants have a type rule, or more precisely, a type axiom because there are no premises. The types of all the constants in CONTROL are given in Appendix A from which the corresponding type rules can be inferred. Some rules, such as the rule for $\mathtt{if}_\theta$, are axiom schemas which define a family of constants, in this case an `if` function for each type.

The lift0 typing rule has a side condition, $x \in NonBeh$. This is required to ensure that the argument to `lift0` is not a behaviour type, as discussed in Section 5.1.

The rules for `$*`, `until-then`, and `integral` are straightforward. The var, λ, app and μ rules are standard for PCF-like languages (see [Rey98, pp. 319]). The construct ν is the recursive binding mechanism for behaviours from Chapter 6.

8.3 Explicit typing

We take an intrinsic view of types, which means that only terms that have a valid typing judgement have any meaning, and that the meaning of a term may depend on the typing judgement, not just on the term itself. In general, if the meaning of a term may depend on its typing judgement, then terms with many different typing judgements may not have a unique meaning. However, we shall see that given a valid context there is a unique typing judgement for every term, which allows us to omit typing information in semantic definitions. We will prove this and also give a simple algorithm for type checking terms.

The syntax requires a type annotation for all bound variables. This uses

var	$\dfrac{}{\Gamma \vdash x : \Gamma x}$	
λ	$\dfrac{[\Gamma \vert x : \phi] \vdash F : \theta}{\Gamma \vdash \lambda x{:}\phi\,.\,F : \phi\texttt{->}\theta}$	
app	$\dfrac{E : \phi\texttt{->}\theta \quad F : \phi}{E\,F : \theta}$	
μ	$\dfrac{[\Gamma \vert f : \theta] \vdash F : \theta}{\Gamma \vdash \mu f{:}\theta\,.\,F : \theta}$	
lift0	$\dfrac{x : \theta}{\texttt{lift0}\ x : \texttt{Beh}\ \theta}$	$x \in NonBeh$
$*	$\dfrac{FB : \texttt{Beh}\ (\phi\texttt{->}\theta) \quad A : \texttt{Beh}\ \phi}{FB\ \texttt{\$*}\ A : \texttt{Beh}\ \theta}$	
until-then	$\dfrac{B : \texttt{Beh}\ \theta \quad C : \texttt{Beh Bool} \quad D : \texttt{Beh}\ \theta}{B\ \texttt{until}\ C\ \texttt{then}\ D : \texttt{Beh}\ \theta}$	
integral	$\dfrac{A : \texttt{Beh Real}}{\texttt{integral}\ A : \texttt{Beh Real}}$	
ν	$\dfrac{[\Gamma \vert a : \texttt{Beh}\ \theta] \vdash A : \texttt{Beh}\ \theta}{\Gamma \vdash \nu a{:}\texttt{Beh}\ \theta\,.\,A : \texttt{Beh}\ \theta}$	

Figure 8.1: Typing rules

the notation ⟨Variable⟩ : ⟨Type⟩ which is similar to the notation ⟨Term⟩ : ⟨Type⟩ that appears in the type rules for expressing the types of terms. It is not necessary for these notations to be the same, and to be clear we will emphasise the distinction: the only purpose of type annotations in terms is to simplify the process of deriving typing judgements, whereas a typing judgement provides all the type information necessary to interpret a term. So the process of deriving a valid typing judgement for a term is a necessary part of interpreting the term semantically. However, the process is completely routine, because with the help of the type annotations on bound variables it is possible to obtain a valid typing judgement using a simple bottom up approach. This is what is meant by explicit typing. Terms could contain more typing information—every subterm could be explicitly annotated by its type—but this would be tiresome and make programs difficult to read. Annotating bound variables is the minimum typing information that must be present, for arbitrary terms, to enable explicit typing.

The bottom up type checking algorithm is given in Figure 8.2. It works by recursively traversing the syntatic structure of the term and constructing the type bottom up. This mirrors the way we would construct a typing derivation using the rules. We begin with a term E belonging to the grammar and a valid context Γ. The type rules are syntax directed, that is, there is one rule for each syntactic construct, and this rule defines the type of a term from the types of its immediate subterms. Therefore there must be one type rule that matches the top-level syntactic construct for E, and so on recursively until we reach the leaves (terminals of the grammar). So, if the leaves have unique types then by induction all finite terms have unique types. The leaves are either constants, which have a fixed type, or variables, which are either free or bound in E. If a variable is free in E then Γ gives its type. If it is bound then at some point in the derivation tree the type of the variable is provided by a type annotation, and this will have been added to the context by the rule for the binding construct. Thus the types of all terminals are known and the type of the overall term is built up compositionally from these types. Type checking would be more difficult without explicit types for bound variables, and in fact many programs would have more than one valid typing judgement (e.g., $\lambda x.x : \theta\texttt{->}\theta$ for any θ). This is not the case with explicit typing, as we shall now prove.

Theorem 8.1 (Uniqueness of typing judgements) *For any context Γ that is valid for a term E, there is either a unique typing judgement of the form*

$$\Gamma \vdash E : \theta,$$

or else there is no valid typing judgement for E in Γ.

Proof By induction on the structure of proofs of typing judgements. For each typing rule we assume that the property holds for the premises and we show that it holds for the conclusion. If there are no judgements satisfying all the premises then there is no valid judgement for the overall term, so the theorem holds in such cases.

Base cases.

All constants have a unique type by definition.

var: The context gives the unique type for any variable.

Inductive cases.

λ: If there is a judgement $\Gamma \vdash F : \theta$, then the conclusion $\Gamma \vdash \lambda x : \phi.F : \phi \to \theta$ is unique because ϕ is fixed by the type annotation on x, and θ is unique by the induction hypothesis.

$T : Context \times Term \rightharpoonup Type$

$$T(\Gamma, \mathtt{0}) = \mathtt{Real}$$

(and similarly for all constants),

$$T(\Gamma, x) = \Gamma x$$

$$T(\Gamma, \lambda x : \theta.E) = T([\Gamma | x : \theta], E)$$

$$T(\Gamma, E\ F) = \left\{ \begin{array}{l|l} \theta & T(\Gamma, E) = \phi\texttt{->}\theta \\ & T(\Gamma, F) = \phi \\ error & \end{array} \right.$$

$$T(\Gamma, \mu x : \theta.E) = \left\{ \begin{array}{l|l} \theta & T([\Gamma | x : \theta], E) = \theta \\ error & \end{array} \right.$$

$$T(\Gamma, \mathtt{lift0}\ \ x) = \left\{ \begin{array}{l|l} \mathtt{Beh}\ \ T(\Gamma, x) & x \in NonBeh \\ error & \end{array} \right.$$

$$T(\Gamma, E\mathtt{\$*}F) = \left\{ \begin{array}{l|l} \mathtt{Beh}\ \ \theta & T(\Gamma, E) = \mathtt{Beh}\ \ (\phi\texttt{->}\theta) \\ & T(\Gamma, F) = \mathtt{Beh}\ \ \phi \\ error & \end{array} \right.$$

$$T(\Gamma, B\ \mathtt{until}\ \ C\ \mathtt{then}\ \ D) = \left\{ \begin{array}{l|l} \mathtt{Beh}\ \ \theta & T(\Gamma, B) = T(\Gamma, D) = \mathtt{Beh}\ \ \theta \\ & T(\Gamma, C) = \mathtt{Beh}\ \ \mathtt{Bool} \\ error & \end{array} \right.$$

$$T(\Gamma, \mathtt{integral}\ \ A) = \left\{ \begin{array}{l|l} \mathtt{Beh}\ \ \mathtt{Real} & T(\Gamma, A) = \mathtt{Beh}\ \ \mathtt{Real} \\ error & \end{array} \right.$$

$$T(\Gamma, \nu x : \theta.E) = \left\{ \begin{array}{l|l} \mathtt{Beh}\ \ \theta & T([\Gamma | x : \theta], E) = \mathtt{Beh}\ \ \theta \\ error & \end{array} \right.$$

Figure 8.2: A bottom up type checking algorithm

app: If there are valid judgements $\Gamma \vdash E : \phi\texttt{->}\theta$ and $\Gamma \vdash F : \phi$, then they are unique by the induction hypothesis, and so $\Gamma \vdash EF : \theta$ is the unique judgement.

Similarly for all remaining rules: if there are unique judgements for the premises, then they will construct a unique judgement in the conclusion. □

COROLLARY 8.2 *For any closed term E, there is a unique typing judgement of the form*

$$\vdash E : \theta. \qquad \square$$

8.4 Semantics of non-behaviour terms

Terms in the grammar for NonBeh do not use behaviours at all, and these terms have a particularly simple meaning because they are terms of the simply typed lambda calculus with a recursion operator. We need a denotational semantics for these terms because non-behaviour values can be lifted using `lift0`, and then used in conditions of reactive behaviours. Conditions must be evaluated to find when behaviours react, and so their value must be known.

As for arbitrary terms, non-behaviour terms only have a meaning if they satisfy a typing judgement. Valid typing judgements are specified by exactly the same inference rules as for arbitrary terms, but because none of the constructs in the syntax build behaviours all terms will be of non-behaviour type.

The domains for each (non-behaviour) type are as discussed in Section 4.2; `Real` and `Bool` correspond to flat domains and functions correspond to ω-continuous functions between pointed ω-CPOs. This ensures that the least fixed point theorem can be applied to interpret recursive definitions. Figure 8.3 gives these domains and the semantic function F for non-behaviour terms.

The semantic function F is valid for open and closed terms. It is useful to be able to interpret open terms so that we can reason about program fragments and programs parameterised by a variable. To interpret open terms we need an environment that maps variables to values:

Definition An environment $u \in \mathrm{Env}$ is a partial function from variables to values:

$$\mathrm{Env} = \mathrm{Variable} \rightharpoonup \mathrm{Value}. \qquad \square$$

An environment u is valid for a context Γ if it assigns meanings to values in the appropriate domains for all variables; that is,

$$\mathrm{dom}\ u \supseteq \mathrm{dom}\ \Gamma \ \wedge\ \forall v \in \Gamma, uv \in [\![\Gamma v]\!].$$

Here $[\![\Gamma v]\!]$ is the domain corresponding to the type of v (the context Γ maps variables to types).

To interpret a term E we require a valid context, Γ (which can be empty if E is closed), and an environment that is valid for Γ. This ensures that the semantic equations are type correct. For terms of type θ

$$F[\![\_]\!] \in \mathrm{Term} \to \mathrm{Env} \to [\![\theta]\!].$$

The semantic equations (in Figure 8.3) for constants and variables are straightforward. Lambda abstractions build continuous functions (for a proof of this see [Gun92, pp. 130]). Applications evaluate the function and argument in the environment and then use function application. Finally, recursive definitions compute least fixed points which are guaranteed to exist by the

Types: $\theta ::= \texttt{Real} \mid \texttt{Bool} \mid \theta \texttt{ -> } \theta$

$$
\begin{aligned}
[\![\texttt{Real}]\!] &= \mathbb{R}_\perp \\
[\![\texttt{Bool}]\!] &= \mathbb{B}_\perp \\
[\![\theta_1 \texttt{ -> } \theta_2]\!] &= [\![\theta_1]\!] \to [\![\theta_2]\!]
\end{aligned}
$$

Terms: $K \in Constants,\ x \in Variables$

$E ::= K \mid x \mid \lambda x{:}\theta.E \mid E\ E \mid \mu x.E$

$$
\begin{aligned}
F[\![\texttt{0}]\!]u &= 0 \qquad (etc.) \\
F[\![x]\!]u &= u(x) \\
F[\![\lambda\texttt{x}:\theta\,.\,L]\!]u &= d \mapsto F[\![L]\!][u|x:d] \\
F[\![M\ \ N]\!]u &= (F[\![M]\!]u)(F[\![N]\!]u) \\
F[\![\mu\texttt{x}:\theta\,.\,L]\!]u &= \mathrm{fix}_\theta(d \mapsto F[\![L]\!][u|x:d])
\end{aligned}
$$

$$\text{where } \mathrm{fix}_\theta(f) = \textstyle\bigsqcup_{n=0}^{\infty} f^n(\perp_\theta)$$

Figure 8.3: Direct denotational semantics of non-behaviour terms

least fixed point theorem (and they correspond to the functions we expect from a computational perspective—that is, "unwinding" recursive functions). These semantic equations give the unique meaning of any well-typed non-behaviour term.

8.5 Substitution

We will define substitution which is required for the evaluation rules that are used in our semantics. We then give some properties of the semantic function F with regard to substitution.

It is easier to define the simultaneous substitution of all free variables in an expression and then take the substitution of a single variable as a special case. This requires a substitution map that gives the terms to substitute for each free variable:

DEFINITION A substitution map is a partial function from Variables to Terms,

$$\delta \in \text{Variable} \rightharpoonup \text{Term}. \qquad \square$$

A substitution map δ is valid for a term E in the context Γ if it assigns terms to all the free variables in E,

$$\text{dom } \delta \supseteq FV(E),$$

and if each term is the same type as the variable it replaces,

$$\forall v \in \text{dom } \delta, \Gamma \vdash \delta v : \Gamma v.$$

This requires Γ to be a context that gives the types of all the free variables in E and all the free variables in the terms in the range of δ.

One subtlety of substitution is that we must be careful that free variables in the terms we are inserting are not bound by mistake. This can happen when we substitute open terms into the body of a λ-abstraction or other binding mechanism. The solution we adopt is to rename bound variables to some completely fresh variable so that this problem cannot arise. This raises a slight complication, however, because unless you have some canonical way of choosing new variables, this method can lead to different terms resulting from the same substitutions. In practice, this does not cause any real difficulty because such terms are semantically equivalent, they just have different names for certain bound variables. Therefore it is usual to study terms modulo renaming of bound variables. This is known as α-equivalence in the λ-calculus.

(It is often overlooked that name clashes do not arise when evaluating programs, that is, closed terms. It is easy to show that evaluation—by which we mean leftmost outermost reduction, stopping at λ-abstractions, as in Section 8.6—never substitutes open terms if the overall term is closed, and so substitution can be simplified by removing renaming entirely.)

Definition Substitution of all the free variables of an expression E by the substitution map δ is written E/δ and defined by the following equations:

$$0/\delta = 0$$

(and similarly for all constants),

$$x/\delta = \delta x$$

$$\lambda x:\theta\,.\,E/\delta = \lambda x_{new}:\theta\,.\,(E/[\delta|x : x_{new}])$$

$$\text{where } \forall w \in FV(E) \setminus \{x\} : x_{new} \notin FV(\delta w)$$

(and identically for μ and ν in place of λ),

$$(E\ \ F)/\delta = (E/\delta)(F/\delta)$$

(and / distributes through all the remaining operators). $\square$

Substitution of a term N for a single variable x in E can then be achieved by

$$E/[id_{Var}|x : N]$$

where id_{Var} is the identity function on variables. This gives the substitution that maps all variables to the same variable, except for x which is mapped to the term N. This notation is slightly cumbersome, so for substitution only we will interpret the notation $[x : N]$ as $[id_{Var}|x : N]$.

It is essential that substitution preserves types. This can be proved by straightforward induction on the above definition of substitution.

THEOREM 8.3 (SUBSTITUTION PRESERVES TYPES) *If Γ is a valid context for both E and δ, and δ is a valid substitution map for E, then*

$$\Gamma \vdash E : \theta \implies \Gamma \vdash E/\delta : \theta \qquad \square$$

Next we consider some properties of the semantics of non-behaviour terms (i.e., F) with regards to substitution and environments. In the following theorems we assume that all environments and substitution maps are valid for the terms involved. Proofs of these properties can be found in Reynolds book [Rey98].

THEOREM 8.4 (COINCIDENCE THEOREM FOR F) *If $ux = u'x$ for all $x \in FV(E)$, then $F[\![E]\!]u = F[\![E]\!]u'$.* $\square$

THEOREM 8.5 (SUBSTITUTION THEOREM FOR F) *If* $ux = F[\![\delta x]\!]u'$ *for all* $x \in FV(E)$ *then* $F[\![E]\!]u = F[\![E/\delta]\!]u'$. □

THEOREM 8.6 (RENAMING THEOREM FOR F) *If* $x_{new} \notin FV(E)\backslash\{x\}$*, then*

$$F[\![\lambda x : \theta.E]\!] = F[\![\lambda x_{new} : \theta.(E/[x : x_{new}]]\!].$$

(and similarly for μ). □

8.6 Evaluation rules

The reduce transition rule—which will be defined in the next section—performs evaluation on terms to reduce function applications and expand recursive functions. We use the terminology evaluation rather than reduction because evaluation stops at λ-abstractions; that is, we never evaluate inside an abstraction. The evaluation rules are defined as a relation on terms as follows:

DEFINITION A term E evaluates in one step to E' if (E, E') belongs to the relation $\rightarrow \in Term \times Term$, which is defined by the axiom schemas:

$$\beta \qquad (\lambda x{:}\theta\,.\,L)\,\mathrm{N} \;\rightarrow\; L/[\mathrm{x{:}N}]$$

$$\mu \qquad (\mu x{:}\theta\,.\,L) \;\rightarrow\; L/[\mathrm{x{:}}(\mu x{:}\theta\,.\,L)]$$

and the inference rule

$$\text{lazy} \qquad \frac{M \rightarrow M'}{M\;\;N \rightarrow M'\;\;N}. \qquad \Box$$

These three rules are standard for an operational semantics of PCF-like languages; see [Ten91, pp. 104] or [Gun92, pp. 106]. However, we need to perform evaluation on behaviour valued terms, so we will prove that some standard properties which hold for the PCF fragment also hold for the complete language.

The $\rightarrow$ relation is deterministic, which is equivalent to stating that it is a partial function. We will now prove this property.

THEOREM 8.7 (EVALUATION IS DETERMINISTIC) *For any term E, if $E \rightarrow E'$ and $E \rightarrow E''$ then $E' \equiv E''$. Consequently, $\rightarrow$ is a partial function.*

PROOF By induction on the structure of terms. We will show that for any term only one evaluation rule can apply, and it always gives a unique evaluation step. There are two cases, because the evaluation rules can only be applied to an application or a recursive definition.

Case $E \equiv MN$. If $M \equiv \lambda x : \theta.L$ then the β rule applies (the lazy rule cannot apply because an abstraction cannot be evaluated further, that is, there does not exist M' such that $M \rightarrow M'$) and otherwise the lazy rule applies. The result of the β rule is a substitution, which is unique assuming a canonical choice of new variable names. For the lazy rule, M' is unique by the induction hypothesis, and so the result $M'N$ is unique.

Case $E \equiv \mu x : \theta.L$. The μ rule gives a unique term, again assuming a canonical choice of new variables in the substitution. □

As we would expect, evaluation preserves closedness.

THEOREM 8.8 *If E is closed and $E \rightarrow E'$ then E' is closed.*

PROOF By induction on the structure of terms.

Case $E \equiv (\lambda x : \theta.L)N$. The β rule applies. The overall term is closed and so N must be closed. The only free variable in L is possibly x, so if the closed term N is substituted for x in L, then the resulting term is closed.

Case $E \equiv (\mu x : \theta.L)$. The μ rule applies. x is the only possible free variable in L, so if it is substituted for the term $\mu x : \theta.L$, which is closed by assumption, then the result is closed.

Case $E \equiv MN$. The lazy rule applies. M and N must be closed because E is closed. M' is closed by the induction hypothesis, and therefore $M'N$ is closed. $\square$

This property allows us to evaluate programs by performing sequences of evaluation steps because a closed term will always remain closed. Let $\rightarrow^*$ be the transitive, reflexive closure of $\rightarrow$, then $E \rightarrow^* E'$ signifies that E evaluates to E' in a finite number of steps (possibly zero). This relation is not a function, but because $\rightarrow$ is a function there is only one term that E can evaluate to in any given number of steps.

An important property of evaluation is that it preserves types, which we state formally in the following subject reduction theorem.

THEOREM 8.9 (SUBJECT REDUCTION FOR $\rightarrow$) *If* $\Gamma \vdash E : \theta$ *and* $E \rightarrow E'$ *then* $\Gamma \vdash E' : \theta$.

PROOF By induction on the structure of terms.

Case $E \equiv (\lambda x : \phi.L)N$. The β rule applies. By assumption, $\Gamma \vdash (\lambda x : \phi.L)N : \theta$. This judgement must be obtained from the app rule, hence the premises

$$\Gamma \vdash \lambda x : \phi.L : \phi \rightarrow \theta \text{ and } \Gamma \vdash N : \phi$$

must be valid. Then, by the λ type rule we must have

$$[\Gamma|x : \phi] \vdash L : \theta$$

and so $L/[x : N]$ is a valid substitution because both x and N have type ϕ. Therefore, by Theorem 8.3, $\Gamma \vdash L/[x : N] : \theta$ as required.

Case $E \equiv (\mu x : \theta.L)$. The μ rule applies. By assumption

$$\Gamma \vdash \mu x : \theta.L : \theta$$

and by the μ-rule the premis

$$[\Gamma|x : \theta] \vdash L : \theta$$

must hold. Hence $L/[x : (\mu x : \theta.L)]$ is a valid substitution because both x and $\mu x : \theta.L$ have type θ, and by Theorem 8.3 $\Gamma \vdash L/[x : (\mu x : \theta.L)] : \theta$.

Case $E \equiv MN$. The lazy rule applies. By the induction hypothesis M' has the same type as M, and so by the app rule $M'N$ must have the same type as MN. $\Box$

We will need this theorem to prove subject reduction for our transition system, and ultimately soundness of the type system with respect to evaluation (i.e., well typed programs will not go wrong with a type error at any stage during evaluation).

8.7 Transition rules

In Section 6.5 we motivated a transition system to formalise the operational method of non-reactive evaluation. Here we give the complete rules for the transition system and prove some important properties.

Recall that the meaning of a behaviour depends on the set of times when it is alive. To interpret a behaviour over consecutive intervals using transitions we start with a (term, set of times) pair and make a transition to another such pair. For example, the term

```
1 until (time >= 1) then 2
```

alive for times in $\mathbb{T}$ makes a transition to the term 2 alive for times in $[1, \infty)$. The transition rules also specify the value of the behaviour over the interval, so one possibility is to define the transition system as a ternary relation,

$$\longrightarrow \;\; \in \;\; (Term \times \mathbb{P}(\mathbb{T})) \times Value \times (Term \times \mathbb{P}(\mathbb{T})).$$

For readability, we will write an element of the relation such as

$$((\texttt{1 until (time>=1) then 2}, \mathbb{T}), t \mapsto 1, (2, [1, \infty))) \quad \in \quad \longrightarrow$$

using the notation

$$\texttt{1 until (time>=1) then 2} \xrightarrow[\mathbb{T} \setminus [1,\infty)]{t \mapsto 1} \texttt{2}.$$

This emphases that the value of the behaviour over the interval $\mathbb{T} \setminus [0, \infty)$ is $t \mapsto 1$.

Sometimes we can interpret a behaviour for all times. For example, the behaviour `lift0 1` represents $t \mapsto 1$ for all times. In such cases it does not make sense for the transition rule to give a new (term, set of times) pair because there are no more transitions that can be made. In other words, such transitions have reached a terminal configuration. To make this clear, we write the empty term, ε, as the resulting term, so the pair $(\varepsilon, \emptyset)$ is a terminal configuration. The empty term is not part of the grammar, so we define the transition system as follows to account for this:

$$\longrightarrow \quad \in \quad (Term \times \mathbb{P}(\mathbb{T})) \times Value \times (Term \cup \{\varepsilon\} \times \mathbb{P}(\mathbb{T})).$$

The transition rules are given in Figures 8.4, 8.5 and 8.6. In the rules for \$*, integral and ν there is a problem when the premis transition yields ε because the conclusion transition will construct a new behaviour using ε, but ε is not in the grammar. However, this is a minor point because in such cases the value of the overall behaviour is known for all times, so the new behaviour is not required. One way to address this problem is to assert that whenever ε appears in any premise, the resulting value in the conclusion is always ε.

Recall that to interpret open terms we require an environment, u, and that we can add this to the transition rules using the notation for assumptions lift0

$$\frac{}{u \vdash \texttt{lift0}\ E \xrightarrow[T_0 \setminus \emptyset]{t \mapsto F[\![E]\!]u} \varepsilon}$$

$*

$$\frac{F \xrightarrow[T_0 \setminus M]{f} F' \quad B \xrightarrow[T_0 \setminus M]{b} B'}{F\ \texttt{\$*}\ B \xrightarrow[T_0 \setminus M]{t \mapsto (f(t))(b(t))} F'\ \texttt{\$*}\ B'}$$

no-change

$$\frac{B \xrightarrow[T_0 \setminus T_B]{b} B'}{B \xrightarrow[T_0 \setminus X]{b} B} \qquad \begin{array}{l} X \supsetneq T_B \\ X = \uparrow X \end{array}$$

integral

$$\frac{B \xrightarrow[T_0 \setminus M]{b} B'}{\texttt{integral}\ B \xrightarrow[T_0 \setminus M]{I} K\ \texttt{+}\ \texttt{integral}\ B'} \qquad \int b \text{ exists}$$

$$K \equiv Real(\int_{\inf(T_0)}^{\inf(M)} b(s).ds)$$
$$I = t \mapsto \int_{\inf(T_0)}^{t} b(s).ds$$

bad-integral

$$\frac{B \xrightarrow[T_0 \setminus M]{b} B'}{\texttt{integral}\ B \xrightarrow[T_0 \setminus \emptyset]{\perp_{\mathbb{T} \to \mathbb{R}_\perp}} \varepsilon} \qquad \text{no } \int b \text{ exists}$$

Figure 8.4: Transition rules I : Behaviour expressions and no-change

Formulas for occ, non-occ and bad-cond rules:

$$\begin{array}{lcl} T & = & \{t \in T_0 \mid c(t) = true\} \\ Bad & = & \{t \in T_0 \mid c(t) = \bot_{\mathbb{B}}\} \end{array}$$

Transition rules:

occ
$$\frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \texttt{ until } C \texttt{ then } D \xrightarrow[T_0 \setminus \uparrow T]{b} D} \qquad \begin{array}{l} \uparrow T \supseteq M \\ \uparrow T \not\supseteq \uparrow Bad \end{array}$$

non-occ
$$\frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \texttt{ until } C \texttt{ then } D \xrightarrow[T_0 \setminus M]{b} E} \qquad \begin{array}{l} M \not\supseteq \uparrow T \\ M \not\supseteq \uparrow Bad \end{array}$$

$$E \equiv B' \texttt{ until } C' \texttt{ then } D$$

bad-cond
$$\frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \texttt{ until } C \texttt{ then } D \xrightarrow[T_0 \setminus M]{b'} \varepsilon} \qquad \uparrow Bad \supseteq M \cup \uparrow T$$

$$b' = t \mapsto \left\{ \begin{array}{l|l} b(t) & t \notin \uparrow Bad \\ \bot_{[\![\theta]\!]} & \end{array} \right.$$

Figure 8.5: Transition rules II : Reactive behaviours

ν	$\dfrac{[u\|a:x] \vdash B \xrightarrow[T_0 \setminus M]{b} B'}{u \vdash \nu a.B \xrightarrow[T_0 \setminus M]{b} \nu a.B'}$	$x = b$
env	$\dfrac{}{u \vdash a \xrightarrow[T_0 \setminus \emptyset]{u(a)} \varepsilon}$	
reduce	$\dfrac{E \to E'' \quad E'' \xrightarrow[T_0 \setminus M]{e} E'}{E \xrightarrow[T_0 \setminus M]{e} E'}$	

Figure 8.6: Transition rules III: Behaviour definitions and reduce

in inference systems:

$$u \vdash A \xrightarrow[T_0 \setminus M]{a} A'.$$

The environment does not change from one transition to the next because CONTROL is purely declarative. However, it does serve a dual purpose in the transition system; it is also used by the ν rule to bind variables to values.

The lift0 rule yields a constant valued function

$$t \mapsto F[\![E]\!]u$$

using the denotation semantics of non-behaviour terms from Section 8.4.

The rules for `until-then` are exactly as given in Section 6.7 and the rules for `integral` are as given in Section 6.9.

For recursive behaviour definitions the ν rule is like the letbeh rule from Section 6.10 but without the body. This is because `letbeh` is defined as syntactic sugar in terms of λ and ν, as discussed in Section 7.4.

We would like the transition system to be deterministic, in other words, any given (term, set of times) pair should appear at most once (as the first element) in the relation. This makes it possible to assign unique meanings to programs using the transition system, which otherwise would require a proof that different transitions for a term result in the same overall value. To be deterministic we require that only one rule applies to any (term, set of times) pair, and that every rule specifies a unique transition. The first requirement is broken by the no-change rule, which can be used on any (term, set of times) pair, giving two possible rules in most cases. However, in practice the no-change rule is only used when combining sub-behaviours of a compound expression so that all of the behaviours except one can remain unchanged by the overall transition. So long as the overall transition is made over the longest possible interval, the uses of the no-change rule are necessary so there is only one possible derivation of the overall transition. This suggests the following theorem for the determinacy of transitions.

Theorem 8.10 (Transitions are deterministic) *Given any behaviour A, upperset T_0 and valid environment u, there is at most one triple (A', a, M) such that M is the smallest set satisfying*

$$u \vdash A \xrightarrow[T_0 \setminus M]{a} A'.$$

Proof By induction on proofs of transitions. We assume that the theorem holds for sub-proofs and show that the resulting transition is unique.

Case $A \equiv$ `lift0` E. The lift0 rule gives the transition where $A' \equiv \varepsilon$, $a = F[\![E]\!]u$ and $M = \emptyset$. These values are unique (F is a function). Another transition is possible by first applying the no-change rule. In fact, this rule can be applied any number of times before finally applying the lift0 rule. However, any such transition results in larger sets for M (consider the side condition $X \supsetneq T_B$ for the no-change rule). Therefore, the transition with the smallest set M can only be obtained by applying the lift0 rule on its own. The same is true for all other cases, so from now on we will ignore the no-change rule unless it is actually needed in a transition derivation.

Case $A \equiv MN$. The reduce rule is the only rule (other than no-change) which applies. If $A \rightarrow A''$ then A'' is unique (by Theorem 8.7) and by the induction hypothesis

$$A'' \xrightarrow[T_0 \setminus M]{a} A'$$

is the unique transition (for the smallest M).

Case $A \equiv F$ `$*` B. Only the \$* rule applies. By the induction hypothesis, the transitions F and B make, if they can make any, are unique for the smallest sets M_F and M_B. To apply the \$* rule we must apply the no-change rule on one of these transitions so the overall transition can be made over $M = M_F \cup M_B$. This is the smallest such M containing both M_F and M_B (it must contain these sets because the interval of the transition must be non-reactive), and so the overall transition is unique.

The remaining cases are similar to this one. For constructs which have many rules, such as `until-then`, only one applies for any (term, set of times) pair because the side conditions are mutually exclusive. $\Box$

Transitions preserve closedness of terms and preserve types.

THEOREM 8.11 *If A is closed and*

$$u \vdash A \xrightarrow[T_0 \setminus M]{a} A'$$

then A' is closed.

PROOF By induction on proofs of transitions. No transition rule introduces an open term (assuming the induction hypothesis holds) when the initial term is closed, so this proof is straightforward. Note that the case for the reduce rule relies on evaluation preserving closedness. □

THEOREM 8.12 (SUBJECT REDUCTION FOR $\longrightarrow$) *If $\Gamma \vdash A : \texttt{Beh}\ \theta$ and*

$$u \vdash A \xrightarrow[T_0 \setminus M]{a} A'$$

then $\Gamma \vdash A' : \texttt{Beh}\ \theta$. (It is not meaningful to construct a judgement when $A' = \varepsilon$ because ε is not a term, so the theorem does not include this case.)

PROOF By induction on proofs of transitions. The rules lift0, bad-integral, bad-cond and env yield $A' = \varepsilon$ so we do not need to consider them.

Cases \$*, no-change, integral, non-occ and ν. These rules reconstruct the same kind of term with new behaviours, and these new behaviours have the same types as the original ones by the induction hypothesis.

Case occ. The typing rule for `until-then` stipulates that D has the same type as B `until` C `then` D.

Case reduce. By subject reduction for $\rightarrow$, E'' has the same type as E, and by the induction hypothesis E' preserves this type. □

8.8 Semantics of behaviour terms

The semantics of behaviours is defined in terms of the transition system. This gives the meaning of a behaviour term over a non-reactive interval. If a value beyond this interval is required then the next transition must be found, and so on. Thus, if A is a behaviour that satisfies a typing judgement

$$\Gamma \vdash A : \texttt{Beh}\ \theta$$

and u is a valid environment for A, then the meaning of A is given by the following semantic function:

$$B : Term \to \mathbb{P}(\mathbb{T}) \to Env \to \mathbb{T} \to [\![\theta]\!]$$

$$B[\![A]\!](T_0)u = t \mapsto \left\{ \begin{array}{l} a(t) \\ B[\![A']\!](M)u(t) \end{array} \right| \begin{array}{l} t \in T_0 \setminus M \\ \ \end{array}$$

$$\text{where } u \vdash A \xrightarrow[T_0 \setminus M]{a} A'.$$

8.9 Semantics of all terms

Given a term, if it is a behaviour then its meaning can be found using B, and if it is a function in the non-behaviour fragment of CONTROL then its meaning can be found using F. This suggests the following semantic function for a term E satisfying a typing judgement

$$\Gamma \vdash E : \theta$$

and an environment u that is valid with respect to Γ:

$$[\![E]\!]u \quad \in \quad [\![\theta]\!]$$

$$[\![E]\!]u \quad = \quad \left\{ \begin{array}{l} F[\![E]\!]u \\ B[\![E]\!](\mathbb{T})u \end{array} \right| \begin{array}{l} \theta \in \text{NonBeh} \\ \ \end{array}$$

There are other possibilities, however, such as a term which is a function yielding a behaviour, and our semantics does not directly give a meaning for such terms. This is because there are no transitions that a function yielding a behaviour can make—it is necessary to apply the function first. However, because free variables are allowed in our semantics it is possible to apply the function to a variable that is bound within the environment, thus giving a behaviour which can be evaluated with our semantics.

Chapter summary

In this chapter we brought together the methods described in the previous chapters to form a complete formal semantics for CONTROL. We used the transition system introduced in Chapter 6 and the evaluation rules from Chapter 7 to construct an operational semantics for any term. This requires evaluating non-behaviour terms using the standard denotational semantics because to determine which transition to make it is necessary to know the value of condition behaviours.

We proved some useful properties of our semantics. It is deterministic, which requires first proving that the evaluation and transition rules are deterministic. Finally, type soundness follows from the subject reduction theorems.

Chapter 9

Applications of the semantics

The semantics from Chapter 8 can be used to find the value of any well-typed CONTROL program. We will describe how to apply the semantics and then we will provide some examples. For the first three examples we give detailed accounts of how the semantics is used to find the value of each program; for each of the remaining examples we describe the program but do not give a complete interpretation.

9.1 Interpreting programs

We will explain how, in practice, the formal semantics from the previous chapter can be used to find the meaning of any given program.

In some examples we have assumed that `time` is a primitive in CONTROL, but in fact it needs to be defined explicitly. However, we do not want to define `time`, or other often used terms, in every program, so we will assume that there is a 'prelude' which contains such definitions. The first step towards interpreting a program is to add the prelude to the main program. Then we must desugar all `let` statements to obtain a program in the core syntax. Only programs that are well typed—in other words generated by the typed syntax rules—have any meaning; by definition the meaning of programs that are ill typed is undefined. At this stage we have either a program that does not construct any behaviours, in which case we can apply the semantics of non-behaviour terms from Section 8.4, or else we have a program whose result is a behaviour, in which case we use the transition rules. The procedure described so far is shown in Figure 9.1.

Now we will describe the procedure for evaluating a behaviour-valued program. The transition rules yield formulas involving free variables. In turn the free variables are constrained by side conditions, and these side conditions must be solved to find the value of the variables and hence of the overall program. This is generally over a finite interval when no behaviours react, and so this process is repeated to find the meaning of the program for later times. If there are no reactive behaviours in a program, then the rules will give its meaning over all times and no more transitions will be required. For some programs, however, there may always be reactive components, and so this process could continue indefinitely. In such cases, it is necessary to use induction arguments to reason about programs; this is illustrated by the example in Section 9.4. The iterative procedure we have just described is shown in Figure 9.2.

To complete the interpretation, the values obtained over non-reactive intervals are pieced together as follows:

$$[\![P]\!] = t \mapsto \left\{ \begin{array}{l|l} p_0(t) & t \in \mathbb{T} \setminus T_1 \\ p_1(t) & t \in T_1 \setminus T_2 \\ \vdots & \vdots \end{array} \right.$$

where the values p_i and sets T_i are obtained by deriving the transitions

$$P \xrightarrow[\mathbb{T} \setminus T_1]{p_0} P_1 \xrightarrow[T_1 \setminus T_2]{p_1} \dots$$

As we said above, this involves solving equations for all the free variables for each transition.

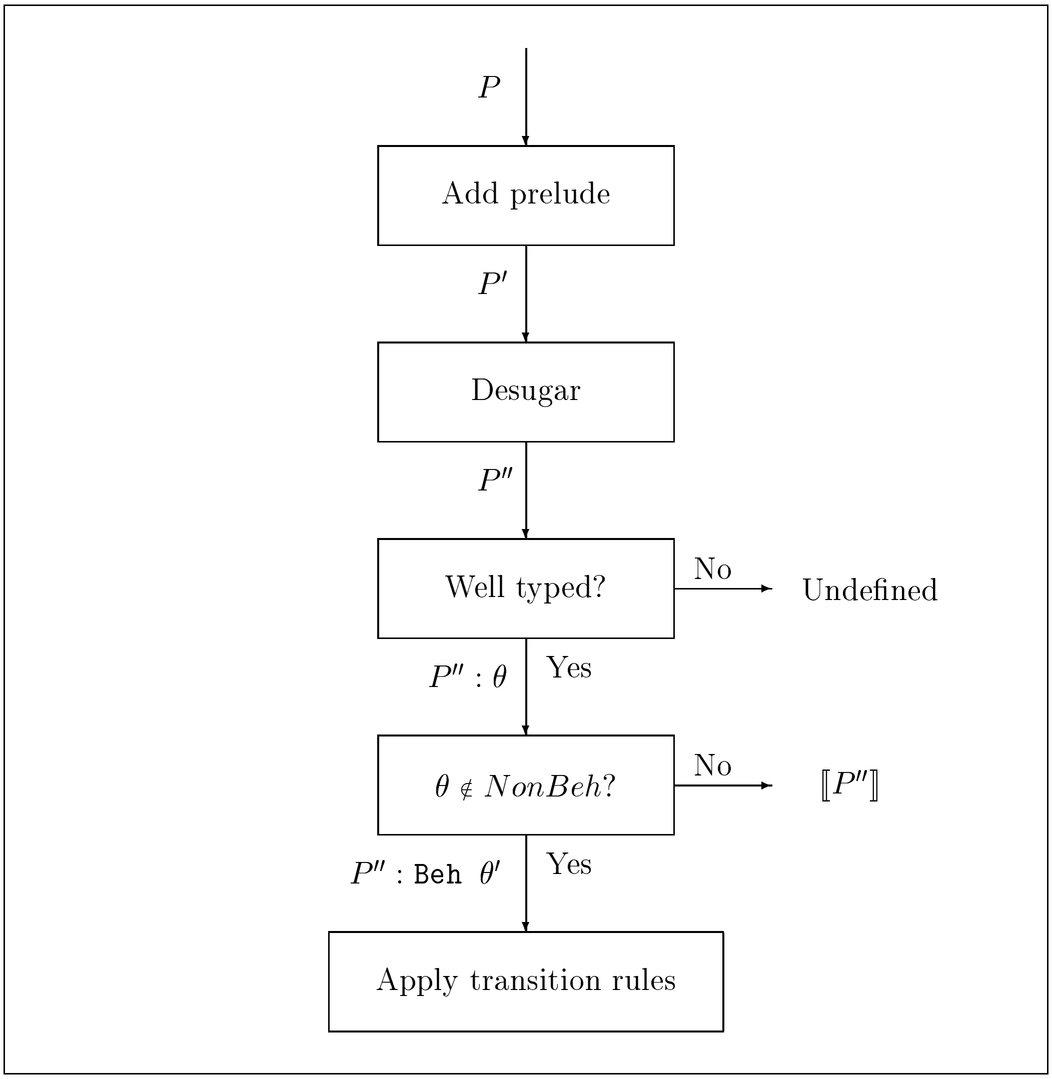


Figure 9.1: Interpreting programs, part I

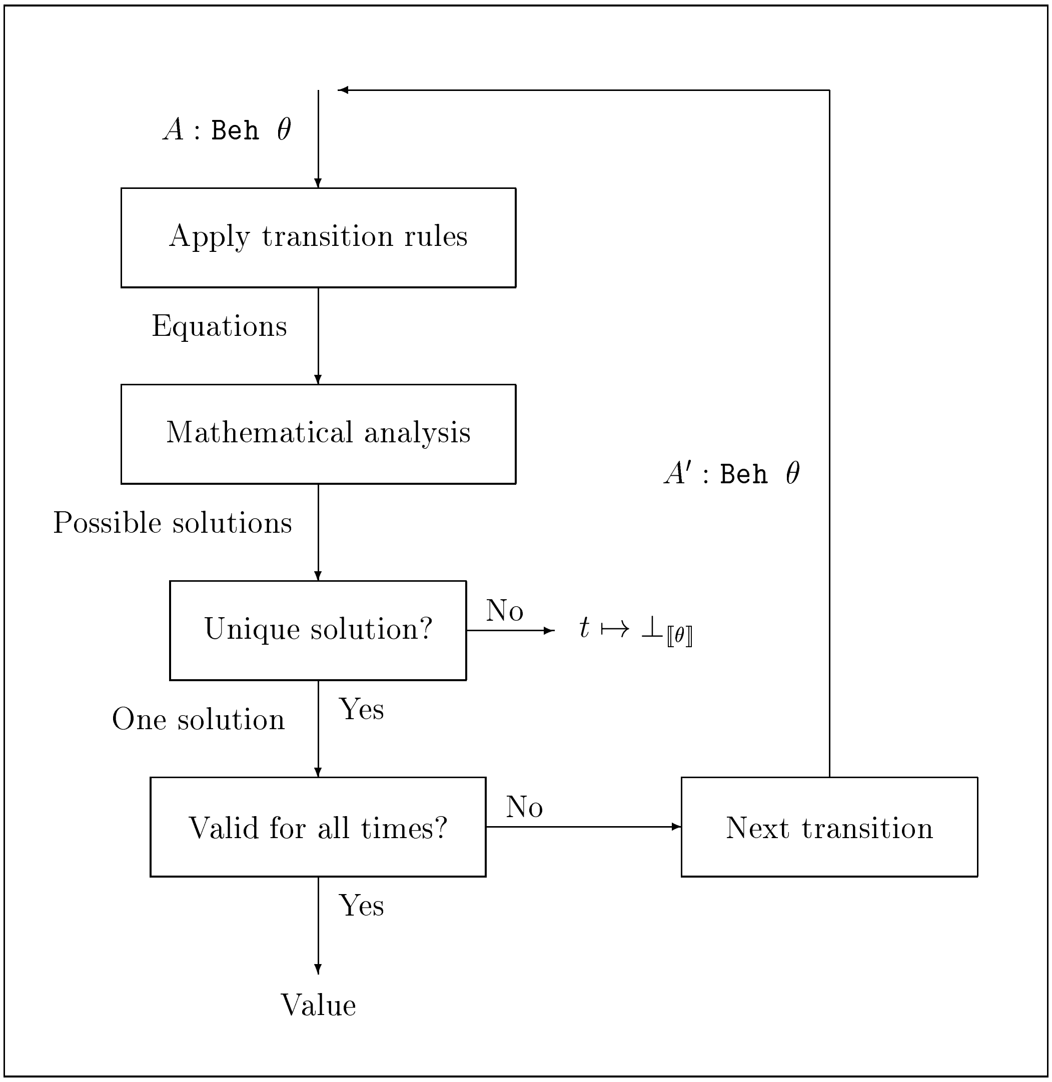


Figure 9.2: Interpreting programs, part II

9.2 A recursive reactive definition

The first example is the program,

```
letbeh a = 1 until (time >= a) then 2 in a.
```

For convenience, we will use B to refer to the right hand side of the definition of a,

$$B \equiv \texttt{1 until (time >= a) then 2}.$$

We begin by de-sugaring the program,

$$\begin{array}{ll} & \texttt{letbeh a = } B \texttt{ in a} \\ \equiv & \langle \text{letbeh} \rangle \\ & (\lambda\texttt{a.a})\ (\nu\texttt{a.}B). \end{array}$$

Let A denote this unsugared program. The next step is to use the transition rules to find the value of A over a non-reactive interval, that is,

$$\vdash A \xrightarrow[\mathbb{T} \setminus M]{a_0} A_1.$$

The term A is an application, and the only rule that applies is the reduce rule which will reduce it to a top-level behaviour, using the evaluation rules, as follows:

$$\begin{array}{ll} & (\lambda\texttt{a.a})\ (\nu\texttt{a.}B) \\ \to & \langle \beta \rangle \\ & \nu\texttt{a.}B. \end{array}$$

Then, by the ν rule, we must interpret B in the environment where a maps to x, that is,

$$[\texttt{a} : x] \vdash B \xrightarrow[\mathbb{T} \setminus M]{b} B', \tag{9.1}$$

$$\dfrac{\dfrac{}{\texttt{1} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto 1} \texttt{1}}\langle\text{lift0'}\rangle \qquad \dfrac{\dfrac{}{\texttt{time} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto t} \texttt{time}}\langle\text{time'}\rangle \quad \dfrac{}{\texttt{a} \xrightarrow[\mathbb{T}\setminus M]{x} \texttt{a}}\langle\text{env'}\rangle}{\texttt{time >= a} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto t\geq x(t)} \texttt{time >= a}}\langle\text{lift2'}\rangle}{[\texttt{a} : x] \vdash \texttt{1 until (time >= a) then 2} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto 1} \texttt{2}}\langle\text{occ}\rangle$$

Figure 9.3: First transition for Example 9.2

and then we solve for

$$x = b \tag{9.2}$$

which is the side condition from the ν rule. The derivation of this transition is best represented as a tree; see Figure 9.3. Note that we use a special convention in tree-like derivations; if the environment is the same in the premis as it is in the conclusion, then we leave it out. The environment does not change at all in this derivation (it is always $[\texttt{a} : x]$), so it only appears at the bottom.

The side condition from the occ rule is,

$$M = \uparrow T, \tag{9.3}$$

where

$$\begin{aligned} T &= \{t \in \mathbb{T} \mid (t \mapsto t \geq x(t))(t)\} \\ &= \{t \in \mathbb{T} \mid t \geq x(t)\}. \end{aligned} \tag{9.4}$$

The derivation tree shows that the variable b used in (9.1) is equal to $t \mapsto 1$. Therefore we can solve side condition (9.2),

$$x = t \mapsto 1$$

and substituting this into side condition (9.4) gives,

$$T = \{t \in \mathbb{T} \mid t \geq 1\} = [1, \infty).$$

In this case, $\uparrow T = T$, and so the variable M in the derivation tree equals $[1, \infty)$ (by side condition (9.3)).

In the derivation tree we made used dashed versions of the rules lift0, time, env and lift2. These are derived rules which combine an application of no-change with the rule so that the transition is over the required interval.

So far we have shown that A represents the function $t \mapsto 1$ over the interval $\mathbb{T} \setminus [1, \infty) = [0, 1)$. The next transition is on the behaviour 2 which is trivial,

$$\frac{}{\mathtt{2} \xrightarrow[{[1,\infty)\ \setminus\ \emptyset}]{t \mapsto 2} \varepsilon}\ \langle lift0 \rangle$$

The $\emptyset$ signifies that there are no more transitions (because the behaviour is not reactive) and so the value is $t \mapsto 2$ for all times in $[1, \infty)$. We now know the complete value of the program,

$$\begin{aligned} [\![\texttt{letbeh a = } B \texttt{ in a}]\!] &= t \mapsto \left\{ \begin{array}{c|l} (t \mapsto 1)(t) & t \in [0,1) \\ (t \mapsto 2)(t) & t \in [1,\infty) \end{array} \right. \\ &= t \mapsto \left\{ \begin{array}{c|l} 1 & t < 1 \\ 2 & \end{array} \right. \end{aligned}$$

9.3 A recursive integral

This example illustrates integral equations. The program is,

```
letbeh a = 1 + integral a in a.
```

Again, we use B to refer to the right hand side of `a`,

$$B \equiv \texttt{1 + integral a}$$

$$\dfrac{\dfrac{}{\mathtt{1} \xrightarrow[\mathbb{T}\setminus\emptyset]{t\mapsto 1} \varepsilon}\langle\text{lift0}\rangle \qquad \dfrac{\dfrac{}{\mathtt{a} \xrightarrow[\mathbb{T}\setminus\emptyset]{x} \varepsilon}\langle\text{env}\rangle}{\mathtt{integral\ a} \xrightarrow[\mathbb{T}\setminus\emptyset]{t\mapsto\int_{\inf(\mathbb{T})}^{t} x(s).ds} \varepsilon}\langle\text{integral}\rangle}{[\mathtt{a} : x] \vdash \mathtt{1\ +\ integral\ a} \xrightarrow[\mathbb{T}\setminus\emptyset]{t\mapsto 1+\int_{\inf(\mathbb{T})}^{t} x(s).ds} \varepsilon}\langle\text{lift2 }(+)\rangle$$

Figure 9.4: First transition for Example 9.3

and then the program is the same as in the previous example except for B. The procedure is therefore the same: de-sugar; use the reduce rule; and find the first transition that B makes; that is, find,

$$[\mathtt{a} : x] \vdash B \xrightarrow[\mathbb{T}\setminus M]{b} B',$$

The derivation tree for this transition is shown in Figure 9.4. Note that there are no side conditions restricting M in the derivation tree, so it can be any set of times. If we choose $M = \emptyset$ then the transition is valid over the interval $\mathbb{T} \setminus \emptyset$; in other words, for all times.

This time the side condition $x = b$ is,

$$x = t \mapsto 1 + \int_{\inf(\mathbb{T})}^{t} x(s).ds$$

and, since $\inf(\mathbb{T}) = 0$, this is the integral equation,

$$x(t) = 1 + \int_{0}^{t} x(s).ds.$$

This integral equation has a unique solution,

$$x_1(t) = e^t.$$

(see [HW91] for details). This is the complete meaning of the program.

9.4 A recursive function

In Section 7.3 we saw the following program

```
letrec  b  =  λn.n until (time >= n+1)
                 then b (n+1)
in b 0.
```

The function `b` takes a number `n` and yields a behaviour that is initially `n` and increments by one for each second after time `n`. The program calls `b` with zero so that the result is a counter starting from zero and incrementing each second.

We will refer to the right hand side of the definition of `b` by B,

$$B \equiv \lambda\texttt{n.n until (time >= n+1) then b (n+1)}.$$

Then we de-sugar and apply the reduce rule as follows:

$$\begin{array}{ll}
 & \texttt{letrec b = B in b 0} \\
\equiv & \langle\text{letrec}\rangle \\
 & \texttt{(}\lambda\texttt{b.b 0) (}\mu\texttt{b.B)} \\
\rightarrow & \langle\beta\rangle \\
 & \texttt{(}\mu\texttt{b.B) 0} \\
\rightarrow & \langle\mu\rangle \\
 & \texttt{(}\lambda\texttt{n.n until (time >= n+1)} \\
 & \quad\texttt{then (}\mu\texttt{b.B) (n+1)) 0} \\
\rightarrow & \langle\beta\rangle \\
 & \texttt{0 until (time >= 0+1) then (}\mu\texttt{b.B) (0+1)}
\end{array}$$

We can apply the transition rules to this last program because it is a behaviour. We will refer to the after-behaviour of this term by F_1;

$$F_1 = \texttt{(}\mu\texttt{b.B) (0+1)}$$

$$\dfrac{\dfrac{}{\texttt{0} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto 0} \texttt{0}}\langle\text{lift0'}\rangle \qquad \dfrac{\dfrac{}{\texttt{time} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto t} \texttt{time}}\langle\text{time'}\rangle \quad \dfrac{}{\texttt{0+1} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto 1} \texttt{0+1}}\langle\text{lift0'}\rangle}{\texttt{time >= 0+1} \xrightarrow[\mathbb{T}\setminus M]{t\mapsto t\geq 1} \texttt{time >= 0+1}}\langle\text{lift2}\rangle}{\texttt{0 until (time >= 0+1) then } F_1 \xrightarrow[\mathbb{T}\setminus[1,\infty)]{t\mapsto 0} F_1}\langle\text{occ}\rangle$$

Figure 9.5: First transition for Example 9.4

The derivation tree for the overall term is shown in Figure 9.5. Notice that this time the environment is empty, because there are no *behaviours* that are defined recursively (`b` is a function and not a behaviour).

The side conditions from the $\langle occ\rangle$ rule are

$$\begin{aligned} M &= \uparrow T \\ T &= \{t \in \mathbb{T} \mid (t \mapsto t \geq 1)(t)\} \\ &= \{t \in \mathbb{T} \mid t \geq 1\} \\ &= [1, \infty). \end{aligned}$$

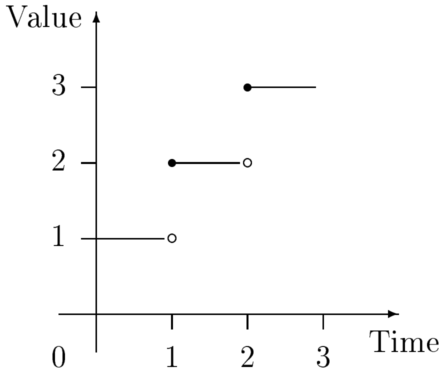
Hence, the first transition tell us that the program means $t \mapsto 0$ over the interval $\mathbb{T} \setminus [1, \infty) = [0, 1)$. The next transition is on the behaviour

$$(\mu\texttt{b.B})\ \texttt{(0+1)}$$

over the interval $[1, \infty)$. However, this is the same as the first transition except with `(0+1)` replacing `0` and $[1, \infty)$ replacing $\mathbb{T}$ throughout. Hence, the transition will yield $t \mapsto 1$ over the interval $[1, 2)$. Then a simple inductive argument shows that the meaning of the program, A, is,

$$[\![A]\!] = t \mapsto \left\{ \begin{array}{c|l} 0 & t \in [0,1) \\ 1 & t \in [1,2) \\ 2 & t \in [2,3) \\ \vdots & \vdots \end{array} \right. .$$

The graph of this function is:



It is now straightforward to show that program A is semantically equivalent to

```
lift0 floor $* time
```

where `floor` is a built in function with the following semantics:

$$[\![\texttt{floor}]\!] = x \mapsto \left\{ \begin{array}{c|l} 0 & x \in [0,1) \\ 1 & x \in [1,2) \\ 2 & x \in [2,3) \\ \vdots & \vdots \end{array} \right. .$$

(i.e., the usual floor function on real numbers.)

9.5 Chess Clocks

A chess clock has two clock faces which show the time each player in a game of chess has remaining. At the start of the game both clocks are set with a fixed amount of time and white's clock begins counting down. After white has moved she presses a button which stops her clock, and black's clock begins to count down. Similarly, after black has moved he presses a button and white's clock again starts to count down. For a computer implementation this requires external input for the buttons, such as mouse button events. CONTROL does not provide such facilities so we will represent button presses by boolean behaviours that are true at times when the button is held down; say `wb` for white's button and `bb` for black's. These behaviours are just free variables in the program, so we can interpret the program with respect to these behaviours.

One way to calculate the time a player has used up is to integrate a playing-indicator behaviour. This is a behaviour that is 1 while a player is taking their turn and 0 while their opponent is. Only one player is using up time at any instant, so when white's playing-indicator switches from 1 to 0 black's should switch from 0 to 1, and vice versa. Therefore it is easier to define the playing-indicators for both players as a pair,

```
letrec pi = (1, 0) until wb then
            (0, 1) until bb then pi
    in ...
```

So we need to extend CONTROL with pairs. The syntax is extended as follows:

$$\begin{array}{lcl} \theta & ::= & \theta{*}\theta \\ E & ::= & (E,\ E) \mid \texttt{fst}\ E \mid \texttt{snd}\ E \end{array}$$

and the type rules for pairs are:

$$\frac{E : \phi \quad F : \theta}{(E,\ F) : \phi{*}\theta} \qquad \frac{E : \phi{*}\theta}{\texttt{fst}\ E : \phi} \qquad \frac{E : \phi{*}\theta}{\texttt{snd}\ E : \theta}$$

The domains for pair types are product domains ordered pointwise, and a pair of values is interpreted denotationally by interpreting each element of the pair.

Returning to the playing indicator `pi`, the semantics of `until-then` ensures that it reacts the next time when `wb` or `bb` is true, ignoring all the button presses in the past. This is exactly what we require. (In Fran this can only be achieved by using user arguments which would complicate the program considerably.) Furthermore, if white presses her button whilst black is playing (or vice versa) it has no effect. This is important because players may accidentally press their button twice in rapid succession.

The behaviour `pi` is a repeating behaviour, so it must be defined by `letrec` and could not be defined by `letbeh`. The amount of time each player has left, say wt and bt, can be defined as follows:

$$wt \equiv \texttt{t0 - integral (fst pi)}$$

$$bt \equiv \texttt{t0 - integral (snd pi)}$$

Here `t0` is the amount of time players have at the start of the game. These values could be represented graphically in an extension of CONTROL with output. (We have implemented a similar program in Fran to display chess clocks.)

Here is a complete program which yields the pair (wt, bt),

```
letrec pi = (1, 0) until wb then
            (0, 1) until bb then pi
    in (t0 - integral (fst pi), t0 - integral (snd pi))
```

Desugaring this program gives,

```
(λ pi :  Beh (Real * Real).
    (t0 - integral (fst pi), t0 - integral (snd pi)))
(μ pi :  Beh (Real * Real).
    (1, 0) until wb then ((0, 1) until bb then pi))
```

Because this program does not make use of recursive behaviours, it can be interpreted by either our denotational or operational semantics. Given values for `wb` and `bb` we can then compute the value of the behaviour. This requires a lengthy but straightforward calculation.

9.6 Water tank

In Section 2.8 we considered a hybrid system that describes a water tank controller which maintains the level of water in a tank by opening and closing a valve; when the level rises to 60 units it closes the valve and when it falls to 30 units it re-opens the valve. This system can be implemented in CONTROL as follows:

```
let beh h = 40 + integral h'

    rec h' =  0.2 until h >= 60 then

             -0.1 until h <= 30 then h'

in h
```

This program is considerably simpler than the description given in the CSP-based specification notation used in Section 2.8. Moreover, it is a CONTROL program and so it is executable as well as a precise specification of the system. Again, to find the complete meaning of the program requires a lengthy but routine calculation. It is then possible to prove conditions such as $30 \leq h \leq 60$, which is the main goal of the final section of He's paper [Jif94].

9.7 Lift

Lifts are a typical example of reactive systems and lift simulations are valuable for analysing proposed designs. Our lift program has only two floors with a button on each floor to call the lift. There are no buttons inside the lift to select which floor.

The buttons are modelled by boolean behaviours `b0` and `b1` for the ground floor and first floor; these are true when the button is pressed and false otherwise. When a button is pressed it lights up and remains lit until the lift stops at that floor. The status of the ground floor button is given by the following behaviour, where `at0` is a boolean behaviour that is true when the lift is at the ground floor,

```
l0 = false until b0  then
     true  until at0 then l0
```

(and similarly for `l1`). So the button is unlit (`l0` is false) until it is pressed (`b0` becomes true) and then it is lit (true) until the lift arrives (`at0`), and then it returns to its original unlit state.

The lift waits at floor 0 until there is a request from floor 1 (i.e., until the button on floor 1 becomes lit). Then it goes up to floor 1 and waits there until there is a request from floor 0. Then it goes down and is back to its initial position. The position of the lift, where `p0` and `p1` are the positions of the floors, is as follows:

```
p = p0     until l1  then
    goUp   until at1 then
    p1     until l0  then
    goDown until at0 then p
```

The position behaviours `goUp` and `goDown` are just linear functions:

```
goUp = p0 + integral v
goDown = p1 - integral v
```

where `v` is the velocity of the lift. (Ideally this should vary as the lift accelerates and decelerates, but as a first approximation it could be constant.)

The behaviour `at0` is just a boolean behaviour:

```
at0 = (p == p0)
```

(and similarly for `at1`).

These definitions seem intuitive but when we try to construct the whole program it is not clear whether we should use `letrec` or `letbeh` to define `p`. It is certainly a repeating behaviour, which suggests that we should use `letrec`, but it also refers to itself (it appears in the conditions `at0` and `at1`) which suggests that we should use `letbeh`. The solution is to define a repeating behaviour `p'` using `letrec`, and then define the actual behaviour `p` in terms of `p'` using `letbeh`. The resulting definitions are mutually recursive so we must use the multiple let statement described in Section 7.7. The same technique is used for `l0` and `l1`. The overall program which gives the position of the lift is as follows:

```
let rec p'  = p0     until l1  then
              goUp   until at1 then
              p1     until l0  then
              goDown until at0 then p'
    beh p   = p'

    rec l0' = false  until b0  then
```

```
                 true   until at0 then l0'
    beh l0  = l0'

    rec l1' = false  until b1  then
              true   until at1 then l1'
    beh l1  = l1'

    beh at0 = (p == p0)
    beh at1 = (p == p0)

in p
```

Chapter 10

Summary and future work

In this chapter we consider the implications of our work and identify the main contributions. Then we describe some possibilities for future work.

10.1 Summary

We have presented a complete formal semantics for a new language called CONTROL. This language provides powerful facilities for describing behaviours and for using them in conjunction with functions. It is intended as a core for practical languages for programming reactive systems, and we believe that such languages will benefit from the simple way temporal and reactive components can be described. This simplicity was demonstrated by the example programs given in Chapter 9.

The semantic theory of CONTROL is interesting firstly because it solves some technical problems concerning events and integration, and secondly because it combines the continuous mathematics of behaviours with the discrete mathematics of functions. Traditionally these branches are quite separate but in the field of reactive systems it is clear that they are both essential. Our theory may therefore have implications to related areas such as specification of reactive systems and hybrid systems.

Our original motivation was to develop a semantics for Fran, and our theory of CONTROL is a valuable first step towards this. Our semantics of the core operators improves on previous attempts by distinguishing events that occur at a given time from those that occur strictly after some time. Also we define concrete domains of values for all types. We will not propose to use our theory as a basis for a semantics for Fran, however, because we have improved the design of CONTROL, based on semantic considerations, and it differs from Fran in a number of ways, as we shall now describe.

Firstly, we have eliminated user arguments (or start times) which complicate reactive programs in Fran. This was achieved by making the times when behaviours are alive implicit in the structure of programs. Secondly, we have introduced a new facility for defining recursive behaviours. This makes the distinction between recursive functions that yield behaviours and actual behaviour objects that have a recursive definition. The examples in this dissertation show the practical importance of this distinction, and the elegant programming style that results. Furthermore, it provides resetable and persistent behaviours which are essential given that the start time of behaviours can no longer be specified.

These changes improve significantly on Fran, and CONTROL programs are often much simpler than the corresponding Fran programs. Consequently we would prefer to change Fran so that its core is based on CONTROL rather than adapting our semantics for the existing Fran language.

Our formal semantics provides a rigorous basis for reasoning about CONTROL programs. For small examples it is possible to use the semantics to prove that programs are correct. This generally involves a combination of techniques; in particular, applying the transition rules, using mathematical analysis for integrals and using induction for recursive definitions. This makes it less suitable for automatic verification compared to many languages, and so we may need to develop new techniques for reasoning about large programs.

We will now consider some specific topics for future work in addition to those mentioned above.

10.2 Implementations of CONTROL

An implementation of CONTROL would allow us to experiment with larger programs and develop the language so that it is usable for real applications. There are a number of difficulties with implementing our semantics, however, as we shall now discuss.

We have developed a theory of an idealised language. In particular, CONTROL makes the following assumptions:

- Real numbers and operations on them are exact.
- Integration of real valued behaviours is exact.
- Events in reactive behaviours are determined exactly.

In practice, we do not have the techniques to implement these features. The operations available on representations of exact real numbers has expanded in recent years ([PEE97]), but not sufficiently to implement CONTROL. The techniques for exact integration ([EE96]) are not enough for CONTROL because integrals may appear in recursive behaviours, and such programs are then equivalent to integral equations rather than plain integrals. However, future progress in this area may allow an implementation of CONTROL using these exact representations of real numbers. That said, it is important to distinguish these techniques from symbolic, or analytic, techniques for solving integral equations. It is highly unlikely that we will ever have symbolic techniques for solving all integral equations because in most cases there does not exist a formula that represents the function that satisfies the equations. More precisely, for some cases it is possible to prove that a function satisfying the equations exists, but that there is no closed form formula for this function.

At present there are two possible approaches for implementations:

1. Restrict the types and operators so that real numbers, integrals and event occurrences can be computed exactly.

2. Use approximation methods.

We have done preliminary investigations into both these approaches. A subset of the reals with exact operations can be obtained using ratios of integers. We have tried this in Haskell by using the type `Ratio Integer` for real numbers. We must be sure to avoid all operations that can compute irrational numbers. One way is to restrict behaviours to linear functions, and consequently integration may only be applied to constant behaviours otherwise the result will be non-linear. Because all behaviours are linear, event detection is simply finding the intersection of straight lines (which are always at rational points). Of course this approach is very restrictive and only useful in applications where we know beforehand that all behaviours are linear.

Approximation methods are used by Fran for the same three features that we require them for in CONTROL. We have experimented with many different representations of behaviours and techniques for integration [Dan97b], but we have not implemented full CONTROL using approximation techniques. One reason is that the embedded language approach that Fran uses is not possible for CONTROL because there are two different mechanisms for recursion. Embedding CONTROL in Haskell would only allow recursive definitions using Haskell's recursion mechanism. For this reason it is necessary to write a new interpreter or compiler for CONTROL rather than use the embedded language approach. Disappointingly for Fran researchers, we cannot improve the existing implementation of Fran with our core language semantics for the same reason.

10.3 Discrete models

It is important to have an operational semantics for approximate implementations. This would provide a formal description for verifying implementations and for reasoning about the behaviour of programs. It must capture the approximation techniques used, ideally in a modular way so that different methods can be substituted into the same semantic framework. A semantics taking this approach will be based on discrete time models of behaviours because the approximation methods are discrete. In this section we will consider discrete models.

A discrete representation of behaviours uses (time, value) pairs to give the value of a behaviour at various points in time. This is similar to imperative streams, except that they also allow side effects [Sch96b]. One way to define a sequence of (time, value) pairs is to define a function which maps lists of times to corresponding lists of values. Using this approach we have defined the core operators of CONTROL as a Haskell program—see Appendix B. It only implements real valued behaviours to illustrate the approach; extending this to all types of behaviours may be possible using the advanced extensions of the Haskell type system [Jon97]. We have used Euler's method [BF93] for calculating integrals.

The discrete model gives an operational semantics for the core operators. It becomes more complicated when λ-abstractions and recursive functions are introduced, but there are no serious difficulties. Recursive behaviours can be accounted for by introducing a variable which denotes a list of values. This will lead to recursively defined lists, but these should be valid for recursive integrals and reactive behaviours for the following reasons:

- Euler's method for integration is designed to solve integral equations, so it does not need the current sample point to calculate the integral at that point, only the previous points. Therefore recursive integrals never create cyclic dependencies.

- Recursive reactive behaviours can be dealt with in the same way Fran does; `until` evaluates the condition at the previous point, ensuring that the definitions are never cyclic.

One interesting idea for validating the semantics of recursive behaviours with this semantics is to use recursive lists in Haskell and then show that they are productive [Sij89].

Given that the discrete model has a relatively simple semantics it seems natural to try and extrapolate an exact semantics as the limit of the approximate semantics. In other words, define the exact semantics to be the list of (time, value) pairs obtained as the gap between points tends to zero. This approach was explored early on in our research, but there are some technical problems:

- Assuming that our aim is to show that discrete models approximate the idealised exact model, the approach is circular because the exact model is in terms of a particular discrete model.

- Say we approach the limit by inserting new points between existing points. At the limit the number (time, value) pairs is countable, and so they do not cover the real line. Therefore it is not a continuous time model.

It may be possible to overcome these problems, but there is another drawback with this approach; it does not provide a convenient and useful theory for reasoning about programs because all values are expressed as limits. This makes it very difficult to manipulate the values obtained, whereas our direct approach gives actual functions of time which are simple to use.

10.4 Approximation and convergence

In the previous section we discussed discrete models of behaviours. The intention is that they approximate the exact semantics and are easier to implement, but we need to establish what it means for a discrete model to approximate the exact semantics. The situation is similar to approximation techniques in numerical analysis, where careful analysis of the errors for each methods is crucial to the development of new methods. The approach taken in numerical analysis is usually the one suggested by Strachey that we mentioned in the introduction: first consider the result if the values were exact, and then consider the errors due to approximation. It may be easier to reason about the errors for individual programs, but if we can obtain useful error results for the language, then we save much repetition of effort.

Following numerical methods the most promising direction is trying to establish when a discrete model converges to the exact semantics as the gap between sample times tends to zero. The idea of convergence is similar to the suggestion in the previous section of extrapolating an exact model from a discrete model, but the problems we identified there are less important here because we are not using convergence to construct a model, merely to establish a correspondence. So convergence is just making a claim about a given discrete model, and if a model converges to the exact semantics then it is likely to be easier to reason about the errors due to approximation than if it does not.

An open question is whether the limit of the discrete model we outlined in Section 10.3 converges to our exact semantics. Without integration we expect there is a close correspondence. The term `lift` x always yields x, and f `$*` b will converge so long as f and b do. Integrals may cause difficulties because our exact semantics yields bottom when there are many solutions to integral equations, whereas Euler's method will always compute a solution. Convergence results for reactivity, functions and recursive behaviours are challenging areas for future work.

Appendix A

Constants in CONTROL

0, 1, ... : Real

true, false : Bool

if$_\theta$: Bool -> θ -> θ -> θ

+, -, *, / : Real -> Real -> Real

-, sin, cos, tan, exp, log : Real -> Real

/\, \/, <-> : Bool -> Bool -> Bool

not : Bool -> Bool

>, <, >=, <= : Real -> Real -> Bool

==$_\theta$, /=$_\theta$: θ -> θ -> Bool

Appendix B

A discrete model of CONTROL in Haskell

```
type MReal = Double
type Time  = MReal
data Beh   = Lift0 MReal
           | Time
           | Lift1 (MReal -> MReal) Beh
           | Until Beh Cond Beh
           | Integral Beh
data Cond  = LiftB (Bool -> Bool -> Bool) Cond Cond
           | LiftC (MReal -> MReal -> Bool) Beh Beh
ats :: Beh -> [Time] -> [MReal]
ats (Lift0 x) ts     = map (\t -> x) ts
ats (Time) ts        = ts
ats (Lift1 f b) ts   = map f (ats b ts)
ats (Until b c d) ts = take i bs ++ ats d (drop i ts)
  where
    bs = ats b ts
    cs = atsC c ts
```

```
    i  = length (takeWhile (== False) cs)
ats (Integral b) ts  = euler ts (ats b ts)
euler ts xs = eulerStep 0 ts xs
  where
    eulerStep s (t0:ts@(t1:_)) (x:xs)
      = s : eulerStep (s + (t1-t0)*x) ts xs
atsC (LiftB op c1 c2) ts
  = zipWith op (atsC c1 ts) (atsC c2 ts)
atsC (LiftC op b1 b2) ts
  = zipWith op (ats b1 ts) (ats b2 ts)
```

References

[Apo74] T. M. Apostol. *Mathematical Analysis.* Addison Wesley, 2 edition, 1974.

[Ber97] G. Berry. The foundations of Esterel, 1997.

[BF93] R.L. Burden and J.D. Faires. *Numerical analysis.* PWS-Kent Publishing Company, 1993.

[Car97] Luca Cardelli. Type systems. Allen B. Tucker (Ed.): The Computer Science and Engineering Handbook. CRC Press, ISBN: 0-8493-2909-4. Chapter 103, pp 2208-2236, 1997.

[CRH93] Zhou Chaochen, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid systems. In *Hybrid Systems, R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (Eds.), LNCS 736, Springer-Verlag*, pages 36–59, 1993.

[Dan84] R. B. Dannenberg. Arctic: A functional language for real-time control. In *ACM Symposium on LISP and Functional Programming*, pages 96–103, 1984.

[Dan97a] Anthony C. Daniels. Fran in action! http://www.cs.nott.ac.uk/˜ acd/publications.html, 1997.

[Dan97b] Anthony C. Daniels. Implementing real-valued functions for integrals and ODEs. http://www.cs.nott.ac.uk/~acd/report.ps, 1997.

[EE96] Martín Escardó and Abbas Edalat. Integration in Real PCF. In *Proceedings of the 11th Annual IEEE Symposium on Logic In Computer Science, New Brunswick, New Jersey, USA*, pages 382–393, 1996.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, 1997.

[Ell96] Conal Elliott. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, 1996.

[Ell97] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *In the Proceedings of the first conference on Domain-Specific Languages*, October 1997.

[Ell98a] Conal Elliott. Composing reactive animations. *Dr. Dobb's Journal*, July 1998.

[Ell98b] Conal Elliott. Declarative event-oriented programming. http://www.research.microsoft.com/ conal/papers/default.htm, 1998.

[Ell98c] Conal Elliott. A fifteen puzzle in Fran. http://www.research.microsoft.com/ conal/papers/default.htm, 1998.

[Ell98d] Conal Elliott. Functional implementations of continuous modeled animation. In *In the Proceedings of PLILP/ALP '98*, 1998.

[Ell98e] Conal Elliott. Two-handed image navigation in Fran. In *1998 Glasgow Functional Programming Workshop*, 1998.

[Ell99a] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *To appear in IEEE Transactions on Software Engineering*, 1999.

[Ell99b] Conal Elliott. From functional animation to sprite-based display. In *Proceedings of PADL '99*, 1999.

[ESAE95] Conal Elliott, Greg Schechter, and Salim Abi-Ezzi. Mediaflow, a framework for distributed integrated media. Technical Report SMLI TR-95-40, Sun Microsystems Laboratories, June 1995.

[ESYAE94] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *Proceedings of SIGGRAPH '94*, 1994.

[Gun92] Carl Gunter. *Semantics of programming languages.* MIT Press, 1992.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings IEEE*, volume 79, pages 1305–1305, 1991.

[Hoa72] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating systems techniques: Proceedings of a seminar. Hoare, C. A. R. and Perrott, R. H. editors. A. P. I. C. studeies in Data processing 9.*, pages 61–71. Academic Press, London, 1972.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice-hall international, London, 1985.

[HW91] J.H. Hubbard and B.H. West. *Differential equations: A dynamical systems approach. Part 1: Ordinary differential equations.* Springer-Verlag, 1991.

[Jif94] He Jifeng. From CSP to hybrid systems. In *A Classical Mind, Essays in Honour of C.A.R. Hoare, Prentice Hall*, pages 171–189, 1994.

[Jon97] Mark P. Jones. First-class polymorphism with type inference. In *In Proceedings of the Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[KM77] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In *IFIP77, B. Gilchrist (ed.), North-Holland*, pages 993–998, 1977.

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems, Kluwer Academic Publishers*, 2(4):255–299, 1990.

[Lan66] P. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMarire. Programming real times applications with Signal. In *Proceedings IEEE*, pages 1321–1336, 1991.

[Lin97] Gary Shu Ling. Fran: Its semantics and existing problems. http://pantheon.yale.edu/˜ sling/research/690Report.ps.zip, 1997.

[Lin98] Gary Ling. Frob—functional robotics. http://www.cs.yale.edu/ ling/, 1998.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Department of Computer Science, University of Edinburgh, 1991.

[Mit96] John C. Mitchell. *Foundations for programming languages.* MIT Press, 1996.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag: Heidelberg, Germany, 1992.

[PEE97] P. J. Potts, A. Edalat, and M. H. Escardó. Semantics of exact real arithmetic. In *Proceedings of the Twelfth Anual IEEE Symposium on Logic in Computer Science*, pages 248–257, 1997.

[PHE99] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Proceedings of PADL '99*, 1999.

[Rey98] John C. Reynolds. *Theories of programming languages.* Cambridge University Press, 1998.

[RR87] G.M. Reed and V.W. Roscoe. Metric spaces as models for real-time concurrency. *Mathematical Foundations of Programming.*

Lec. Notes in Comp. Sci. 298, Springer-Verlag, pages 331–343, 1987.

[Sag98] Meurig Sage. Chronos. http://www.dcs.gla.ac.uk/~meurig/chronos/, 1998.

[Sch90] S. Schneider. *Correctness and communication in real-time systems.* PhD thesis, Oxford University Computing Laboratory, 1990.

[Sch96a] E. Scholz. Pidgets—unifying pictures and widgets in a constraint-based framework for concurrent functional gui programming. In *8th International Symposium on Programming Languages: Implementations Implementations, Logics, and Programs, Aachen, Germany, Springer-Verlag*, 1996.

[Sch96b] Enno Scholz. A monad of imperative streams. In *Glasgow FP workshop*, 1996.

[Sch98] E. Scholz. *A framwork for programming interactive graphics in a functional programming language.* PhD thesis, Freien Universität Berlin, 1998.

[Sco70a] Dana S. Scott. The lattice of flow diagrams. Technical Report PRG-3, Programming Research Group, Oxford University Computing Laboratory, 1970.

[Sco70b] Dana S. Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Programming Research Group, Oxford University Computing Laboratory, 1970.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM journal on computing*, 5(3):522–587, 1976.

[Sco93] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical computer science*, 121, 1993.

[SEYAE94] Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi. Functional 3D graphics in C++—with an object-oriented, multiple dispatching implementation. In *Proceedings of the 4th Eurographics Workshop on Object-Oriented Graphics*, 1994.

[Sij89] Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[SS71] Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG-6, Programming Research Group, Oxford University Computing Laboratory, 1971.

[Sto77] Joseph E. Stoy. *Denotational semantics: The Scott-Strachey approach to programming language theory.* MIT press, 1977.

[Str73] Christopher Strachey. The varieties of programming language. Technical Report Technical monograph PRG-10, Programming research group, Oxford University computing laboratory, 1973.

[Ten91] R. D. Tennent. *Semantics of programming languages.* Prentice-Hall, 1991.

[TF92] Thomas and Finney. *Calculus and analytic geometry.* Addison-Wesley Publishing Company, 1992.

[Tho98] Simon Thompson. A functional reactive animation of a lift using Fran. Technical Report TR 5-98, Computing Laboratory, University of Kent, May 1998.

[Vui90] J. E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, 1990.