Recursive Functions and Reactive Behaviours: The Essence of Fran

Anthony Charles Daniels

University of Kent at Canterbury, Canterbury, Kent CT2 7NF, UK,
`A.C.Daniels@ukc.ac.uk`

Abstract. The functional animation language Fran provides abstract datatypes of "behaviours" to represent time varying values—for example the position of moving objects—and "events" to represent discrete occurrences—for example collisions.
We introduce a small functional language called CONTROL which is designed to capture the essential operations on behaviours and events but in a minimalistic language so that it is easier to define a semantics for our language. Previous semantics for Fran have not explained how functions and behaviours combine, and consequently they cannot interpret functions that yield behaviours. In contrast we provide a complete semantics for CONTROL. This is based on an operational-style transition system for behaviours that allows function terms to be reduced in the usual way.

1 Introduction

Fran [EH97, PEL98] is a Haskell [HJ99, HF92] library for creating interactive animations. It provides constants and operations for two datatypes: behaviours and events. Behaviours are used to describe time-varying, or continuously evolving, values. They may be thought of as abstract representations of functions of time. Events are used to capture user actions, such as mouse clicks, and interaction between behaviours, such as when objects collide. Thus behaviours model continuous change and events model discrete occurrences. Behaviours and events can be mutually dependent: behaviours may react, or change course, upon event occurrences; and events may be determined by Boolean valued behaviours.

An animation using the Fran library is a Haskell program that defines an image-valued behaviour. The system uses a presentation engine which evaluates this behaviour and then creates frames of the animation.

Consider the semantics of Fran programs. Fran is embedded in Haskell so a semantics must begin with a complete semantics for Haskell. However, if we try to add a semantics for behaviours and events, an inevitable conflict arises: these datatypes have an implied semantics derived from their implementation, but this is useless for reasoning about programs because it is at the wrong level of abstraction. We want to capture the essential abstract properties of behaviours and events, and not the details of their implementation in Haskell.

One route is to consider behaviours and events as abstract types and give them a semantics separate from their implementation. Elliott and Hudak adopted this approach[EH97], but their semantics does not correspond to the implemented language because in practice approximation techniques are used to compute behaviours. Furthermore, their semantics does not indicate how these abstract values integrate with Haskell code. Crucially, their work does not account for the semantics of functions that yield behaviours.

For example, consider a step behaviour that yields the value 1 initially and then increments by 1 each second. This can be achieved by writing a function that takes an integer n and gives a behaviour that yields n until a second has elapsed, and then calls itself with $n + 1$. Naively combining Elliott and Hudak's semantics with a denotational semantics of recursive functions requires a CPO for behaviours such that taking fixed points gives the required behaviours. This has not been investigated, and some technical difficulties with this approach are discussed in [Dan99].

Our approach is to study a simple functional language with abstract behaviours built in. Because behaviours are built-in rather than implemented in the language, we can legitimately define their semantics to give the idealised operations. Furthermore, we avoid unnecessary complexity due to the size of Haskell and are still able to address the important semantic issues arising from combining behaviours and events with functions.

The language we present here is called CONTROL—CONtinuous Time Reactive Object Language. It is essentially a simplification of Fran, including only the core operators on behaviours and a minimal functional language. Although it does not contain a datatype for events, it includes reactivity by using Boolean behaviours to represent events. Thus the crucial elements of the Fran paradigm are present, but in a language which has a more manageable semantics.

CONTROL first appeared in [Dan99] and this paper is a concise introduction to a useful subset of the full language. The larger publication defines a strongly typed version (here we consider an untyped language) and includes novel facilities that make the 'start times' (or 'user arguments') in Fran redundant. Related to this is the important distinction made between recursive functions and recursive behaviours; we only consider the former in this paper.

In this paper we give a semantics for CONTROL based on an operational-style transition system which integrates smoothly with the evaluation of function terms. Consequently we are able to interpret functions that yield behaviours. We illustrate our language and its semantics with a simple example that implements chess clocks.

2 Chess Clocks Example

We will introduce CONTROL by describing a program that implements chess clocks. Chess clocks have two clock faces which show the amount of time each player has remaining in a game of chess. At the start of the game both clocks are set with equal amounts of time and white's clock begins to count down. After white has moved they press the white button and black's clock begins to count down, and so on alternately.

Before we describe the CONTROL program we will consider a typical implementation in an imperative language to illustrate the advantages of our approach. The imperative program in Figure 1 uses a loop and a player variable to indicate which player is thinking. In each iteration of the loop some time is subtracted from the current player's time left and their button is checked.

The timing of this program is somewhat confusing. For example, it is not clear whether it is correct to update the clocks and then check for button presses or the other way around. Similarly, we must decide when to get the current time and where to place other parts of the code, such as the code to check if either player has run out of time and the code to draw the clock faces. In practice, the usual assumption is that the loop is performed many times each second and so slight timing irregularities are not significant.

```
timeLeft[white] := initialTime; timeLeft[black] := initialTime;
player := white; t0 := getSystemTime();
loop
  t1 := getSystemTime();
  timeLeft[player] := timeLeft[player] - (t1 - t0);
  if timeLeft[player] <= 0 then exitloop;
  t0 := t1;
  if buttonPress[player] then player := opponent(player);
  // ...code to re-draw the clock faces.
endloop;
```

Fig. 1. Imperative chess clocks

We can implement chess clocks in CONTROL by using a behaviour to capture the time each player has remaining. Suppose we have a behaviour that yields 1 while white is playing and 0 while black is playing; the integral of this behaviour gives the amount of time that white has used up. The equivalent behaviour for black is the opposite, and we define them together as a pair, `p`, as follows:

```
letrec p = lift0 (1, 0) until wb then
           lift0 (0, 1) until bb then p
in       ...
```

Here `p` is a behaviour which toggles between `(1,0)` and `(0,1)` when the buttons are pressed. Initially `p` is `lift0 (1,0)`, which is the constant behaviour that yields `(1,0)` at all times. It switches to `(0,1)` immediately when the boolean behaviour `wb` yields `true`. We use boolean behaviours `wb` and `bb` to model the buttons—they give `true` for times when the appropriate button is held down. The behaviour then restarts when `bb` yields `true`, returning to its initial state. Our `letrec` construct takes the same approach used for recursive definitions as in other normal order functional languages such as PCF [Plo77, Sco93, Mit96]. Intuitively it 'unwinds' the definition as many times as necessary.

The amount of time that white has remaining is the time available at the start, `initialTime`, minus the integral of the first component of `p`,

```
lift0 initialTime - integral (lift1 fst p).
```

The term `lift1 fst p` maps the `fst` function over `p` for all times, so `lift1` is similar to `map` for lists except that behaviours are continuously evolving. Note that the `-` operator is subtraction overloaded for behaviours, defined by applying `lift2` to the standard subtraction operator. The complete program comprises a pair of behaviours giving the time white and black have remaining:

```
letrec p = lift0 (1, 0) until wb then
           lift0 (0, 1) until bb then p
in (lift0 initialTime - integral (lift1 fst p),
    lift0 initialTime - integral (lift1 snd p))
```

One advantage of the CONTROL version is that timing is implicit; no variables for the time are needed and all temporal aspects are dealt with by behaviours. Of course, the implementation of behaviours must address certain timing issues, but the programmer is not burdened with doing so. Consequently, the semantics of CONTROL can be used to verify that programs are correct at a higher level of abstraction than for the imperative program.

A second key advantage of the CONTROL program is that it is modular because behaviours are constructed compositionally. For instance, we can use the pair of behaviours defined above in a program which displays the times on clock faces and checks whether either player has ran out of time. In the imperative program this code has to be inserted into the main loop, resulting in a monolithic block of code. In contrast, the CONTROL program does not need to be changed at all.

3 Language and Semantics

In this section we will describe the syntax and semantics of our language. The syntax has two parts, functional terms,

$$E ::= K \mid x \mid \lambda x . E \mid E\, E \mid \mu x . E \mid (E,\ E) \mid \texttt{fst}\ E \mid \texttt{snd}\ E$$

(K stands for constants such as `0`, `1`, `+`, `-` and `>=`) and behaviour terms,

$$E ::= \texttt{time} \mid \texttt{lift0}\ E \mid E\ \texttt{\$*}\ E \mid \texttt{integral}\ E \mid E\ \texttt{until}\ E\ \texttt{then}\ E$$

We allow functions that yield behaviours. However, only non-behaviour terms may be lifted to constant behaviours using `lift0`. This simplifies our language because it prevents behaviours of behaviours. Of course, this limits the language to a degree, but many interesting programs can be expressed without higher-order behaviours. Although we consider an untyped language here, it is possible to define a strongly typed variant using a minor extension of the simply typed lambda calculus [Dan99].

We will describe these behaviour terms informally in the next two sections and then present our semantics.

3.1 Time, Lifting and Integration

The behaviour `time` yields the current time. Viewed as a function of time it is the identity function $t \mapsto t$. In the chess clocks program we saw the lifting function `lift0` for lifting constants—for example `lift0 (1, 0)`—and `lift1` for lifting functions with one argument—for example `lift1 fst`. There are lifting functions for each arity of function; for example `lift2 (+)` performs pointwise addition for real-valued behaviours. These can be defined in terms of `lift0` and a lifted application operator `$*`, which applies a behaviour yielding functions to a behaviour yielding arguments,

```
lift1 f a = lift0 f $* a
lift2 f a b = lift1 f a $* b
lift3 f a b c = lift2 f a b $* c
```

This allows us to treat all the lifting operators by giving a semantics to `lift0` and `$*`.

To see how this works, it is useful to view behaviours abstractly as functions of time. Then, time, lift0 and \$* correspond precisely to I, K and S combinators [Bar84] as follows:

$$\begin{array}{lll} \text{time} \;\; = t \mapsto t & \leftrightarrow & I\,t \;\;\; = t \\ \text{lift0}\ x = t \mapsto x & \leftrightarrow & K\,x\,t \; = x \\ f\ \$^*\ b = t \mapsto (f\,t)(b\,t) & \leftrightarrow & S\,f\,b\,t = f\,t\,(b\,t). \end{array}$$

The next behaviour operator in the syntax, `integral`, yields the integral of its argument from the start time up to the current time.

3.2 Reactive behaviours

The behaviour operator `until-then` constructs reactive behaviours, that is, behaviours which change course when some event occurs. The general form of an `until-then` term is

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

where C is a Boolean behaviour modelling the event. Such terms act like B until C yields true for the first time, and then act like D forever.

If B, C and D are non-reactive (i.e., do not contain any `until-then` subterms) then this description is fairly clear. However, in general we may have nested `until-then` terms and in particular when D is reactive the meaning is somewhat subtle.

Consider the following nested expression where we omit lifting of numbers and >= to help readability:

$$\texttt{1 until (time >= 1.5) then}\underbrace{\texttt{(2 until (time >= 2.5) then 3)}}_{D_1} \qquad (1)$$

This behaviour should start as the constant behaviour $t \mapsto 1$, and then switch to D_1 at time 1.5. Then it should be the constant behaviour $t \mapsto 2$ until time 2.5 when it should switch to 3. Now consider a slight variation on this example which is the same except for the second condition labelled C_2',

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

Intuitively we expect this behaviour to start as $t \mapsto 1$ and then switch to D_2 at time 1.5 as before. Then it should be $t \mapsto 2$ forever because the condition C_2' will always be false—we have already passed time 0.5 so `(time <= 0.5)` must remain false forever.

The way we capture this interpretation of reactive behaviours is to evaluate all behaviours with respect to a set of times. Conditions in `until-then` expressions are only tested for times in this set. Initially the overall program is evaluated over all times, that is, over the set $\mathbb{T}$. In the preceding example the sub-term D_2 would be evaluated for times in the set $[1.5, \infty)$ because this is when the first `until-then` switched to D_2. The condition inside D_2, labelled C_2', will therefore only be tested for times in this set, and it is false for every such time as required.

3.3 Approach to the Semantics

In this section we will give a high level overview of our semantics. The most difficult operator to capture is `until-then`, so we focus our overview on how reactivity is dealt with. If we view behaviours as functions of time then a reactive behaviour acts like some function, say a_0, until its associated event occurs, and then it acts like another function, say a_1. Considering the whole program there may be many reactive sub-terms and for each event occurrence the behaviour changes to a new function of time. The overall value of the behaviour is the function obtained by piecing together $a_0, a_1, \ldots$, as illustrated in Figure 2.

We capture this 'piecing together' of functions over intervals using a small step operational semantics[Mit96]—whenever an event occurs the reactive term it appears in is simplified. These transition steps depend on the order that events occur, and so it is necessary to calculate when events occur. Recall that in CONTROL we use Boolean behaviours to describe events, so we must find actual functions from times to truth values for all such behaviours so that we can determine when events occur. This leads us to a hybrid approach: an operational semantics that yields denotations at each step, and these values are used to

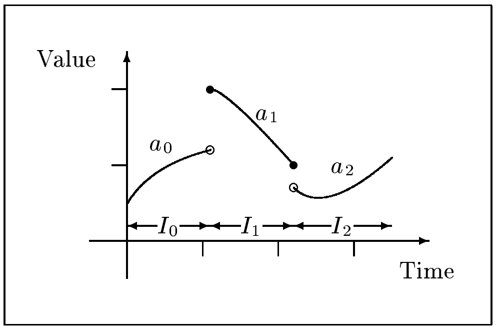

Fig. 2. Piecing together a reactive behaviour

determine the next step. Thus evaluating a behaviour term A yields a chain of values, a_i, over consecutive intervals, I_i. At each step the term is reduced by the operational rules yielding a new term. We write these chains as

$$A \xrightarrow[I_0]{a_0} A_1 \xrightarrow[I_1]{a_1} A_2 \xrightarrow[I_2]{a_2} \ldots$$

The precise meaning of these values is as follows:
A_i is the behaviour term after i transitions
a_i is the mathematical meaning of A_i over the interval I_i
I_i is the longest (possibly infinite) interval over which A_i is non-reactive
To obtain the meaning of A for all times we concatenate these parts,

$$[\![A]\!] = t \mapsto \left\{ \begin{array}{l|l} a_0(t) & t \in I_0 \\ a_1(t) & t \in I_1 \\ \vdots & \vdots \end{array} \right.$$

giving a function like the one illustrated in Figure 2.

3.4 Domains

Our language is untyped, so we interpret terms in a universal domain D^∞ (see [Gun92]). Terms representing real numbers or Boolean values belong to the flat CPOs $\mathbb{R}_\perp = \mathbb{R} \cup \{\perp_\mathbb{R}\}$ and $\mathbb{B}_\perp = \mathbb{B} \cup \{\perp_\mathbb{B}\}$ respectively, where the bottom elements $\perp_\mathbb{R}$ and $\perp_\mathbb{B}$ are necessary to account for non-termination.

The universal domain includes the function space $\mathbb{T} \to D^\infty$ of continuous functions from times to values, where times are positive real numbers; $\mathbb{T} = \{x \in \mathbb{R} \mid x \geq 0\})$. Note that we do not need any conditions on this function space to ensure that functions are continuous because $\mathbb{T}$ has a discrete topology. We do not need a bottom element for times because behaviours are an abstract type and there are no facilities for applying behaviours to times within the language. The space $\mathbb{T} \to D^\infty$ must be a CPO, which it is if we take the least element to be the function which maps all times to $\perp$, and then use a discrete order.

3.5 Transition Rules

Recall that we use transition steps to capture event occurrences. The following transition applies when an event occurs (i.e., when C becomes true),

$$B \texttt{ until } C \texttt{ then } D \xrightarrow[T_0 \setminus \uparrow T]{b} D$$

This asserts that the term B `until` C `then` D makes a transition to the term D, as we would expect when C becomes true. The arrow is decorated with some other values which are as follows:

T_0	is the set of times over which we evaluate the term
T	is the set of times in T_0 when C is true, $\{t \in T_0 \mid c(t) = \mathit{true}\}$
$\uparrow T$	is the upperset of T, $\{s \in \mathbb{R} \mid \exists s' \in T : s' \leq s.\}$
b	is the meaning of B, interpreted over $T_0 \setminus \uparrow T$.

The value on top of the arrow—in this case b—is the value of the term over the interval $T_0 \setminus \uparrow T$, which is why we write the sets of times under the arrow as a set difference. For times after this interval the behaviour will act like D interpreted for times in $\uparrow T$. This is as we described for the nested `until-then` examples—the behaviour D is switched to when C becomes true, so it is evaluated with respect to the set of times at or after any time when C is true, that is, for times in the upperset of T. (Taking the upperset of T captures the fact that the behaviour switches to D permanently.) Taking our earlier example (1) the first transition is:

$$\texttt{1 until (time >= 1.5) then } D_1 \xrightarrow[\mathbb{T} \setminus [1.5,\infty)]{t \mapsto 1} D_1$$

Next D_1 makes the transition

$$\texttt{2 until (time >= 2.5) then 3} \xrightarrow[[1.5,\infty) \setminus [2.5,\infty)]{t \mapsto 2} \texttt{3}$$

Chaining together these two steps gives the value of the behaviour up to time 2.5, and for all later times the behaviour yields `3`.

The rule for transitions like those above is called the occ rule, short for event occurrence. This rule and all the others are given in the Appendix. We will now briefly discuss the remaining rules.

The sub-terms B or C may react before the condition C becomes true, resulting in a transition of the form,

$$B \texttt{ until } C \texttt{ then } D \xrightarrow[T_0 \setminus M]{b} B' \texttt{ until } C' \texttt{ then } D$$

This transition, produced by the non-occ rule, asserts that both B and C make transitions to B' and C'. The no-change rule allows a behaviour to remain the same over a subinterval, and together these rules allow just B to react, just C to react, or both B and C to react simultaneously.

The condition C may yield $\perp$ before it yields true, and in such cases it is impossible to determine when the event occurs. The bad-cond rule captures this case and gives $\perp$ for times after the condition becomes bad. The side conditions for occ, non-occ and bad-cond determine which of the three rules applies to any given `until-then` term. They are mutually exclusive which ensures that transition steps are deterministic.

The rules for lifting and integration are relatively straightforward. The term `lift0` E yields the same value at all times, that is, it equals $t \mapsto [\![E]\!]$ (where $[\![\_]\!]$ is a denotational semantics for non-behaviour terms) over the interval $T_0 \setminus \emptyset$. Because it takes the same value for all times in the future, it never makes a transition. The rule uses the empty term ε to signify this, as do other rules for behaviours that never make another transition. The rule for `integral` A gives the integral of the value of A over non-reactive intervals and accumulates the sum of integrating these non-reactive parts. More detailed descriptions and examples of all these rules are given in [Dan99].

3.6 Transitions for functions

The rule that allows us to combine the semantics of functions and behaviours is the reduce rule,

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

The short arrow $\to$ is a one-step evaluation relation. This rule allows a behaviour to make a transition if it can be evaluated one step and the resulting term can make a transition. Thus it may be applied many times to evaluate a term until it is a behaviour at the top level, and then the transition rule for the appropriate behaviour operator can be applied.

The evaluation relation has three rules: β-reduction,

$$(\beta) \qquad (\lambda x\,.\,L)N \to L[N/x]$$

where $L[N/x]$ means that N replaces x in L; reducing the function in an application (i.e., normal-order evaluation),

$$(\text{norm}) \qquad \frac{M \to M'}{M\ \ N \to M'\ \ N}$$

and unwinding recursive definitions,

$$(\mu) \qquad (\mu x\,.\,L) \to L[(\mu x\,.\,L)/x].$$

To make programs more readable we define syntactic sugar for `let` and `letrec` definitions in the usual way,

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

4 Semantics of Chess Clocks

We will now illustrate our semantics by applying it to the chess clocks program. The interpretation is a direct application of the rules in the Appendix.

The complete chess clocks program yields a pair of real-valued behaviours, so the semantics should give a value from the domain $(\mathbb{T} \to \mathbb{R}_\perp, \mathbb{T} \to \mathbb{R}_\perp)$. In fact, the interesting part is the definition of `p`, so we will start by desugaring the `letrec` definition for `p`,

```
let p = μp. lift0 (1, 0) until wb then
            lift0 (0, 1) until bb then p
in ...
```

Let P be the term on the right hand side of the above definition of `p`. We will deal with P separately, that is, we will construct the chain

$$P \xrightarrow[T_0 \setminus T_1]{p_0} P_1 \xrightarrow[T_1 \setminus T_2]{p_1} P_2 \xrightarrow[T_2 \setminus T_3]{p_2} \dots$$

where $T_0 = \mathbb{T}$, because we begin by evaluating the program over all times, and the sets T_i depend on the button presses.

The term P is a recursive definition so we can unwind it one level using the μ evaluation rule. This gives

$$\begin{array}{l}\texttt{lift0 (1, 0) until wb then} \\ \texttt{lift0 (0, 1) until bb then } P\end{array}$$

In terms of the transition rules, we have used the reduce rule to perform one evaluation step on the term. We now have an `until-then` term at the top level. The occ rule then gives the following transition:

$$\dfrac{\dfrac{\texttt{lift0 (1, 0)} \xrightarrow[\mathbb{T} \setminus \emptyset]{t \mapsto (1,0)} \varepsilon}{\texttt{lift0 (1, 0)} \xrightarrow[\mathbb{T} \setminus T_1]{t \mapsto (1,0)} \texttt{lift0 (1, 0)}}\langle\text{no-change}\rangle \qquad \texttt{wb} \xrightarrow[\mathbb{T} \setminus T_1]{wb} \texttt{wb}}{\texttt{lift0 (1, 0) until wb then } P_1 \xrightarrow[\mathbb{T} \setminus T_1]{t \mapsto (1,0)} P_1}\langle\text{occ}\rangle$$

where

$$\begin{aligned} T_1 &= \uparrow \{t \in \mathbb{T} \mid wb(t) = \mathit{true}\} \\ P_1 &= \texttt{lift0 (0, 1) until bb then } P \end{aligned}$$

Next the behaviour P_1 makes a transition, again by the occ rule. Using a similar derivation to the first transition we obtain:

$$\texttt{lift0 (0, 1) until bb then } P \xrightarrow[T_1 \setminus T_2]{t \mapsto (0,1)} P$$

So far we have found the meaning of P in terms of the first two button presses,

$$[\![P]\!] = t \mapsto \left\{ \begin{array}{l|l} (1,0) & t \in \mathbb{T} \setminus T_1 \\ (0,1) & t \in T_1 \setminus T_2 \\ \ldots & \end{array} \right.$$

The evaluation proceeds by interpreting P over T_2. But P is the term we started with, so the transition for the next interval will be exactly the same except over the set of times T_2 instead of $\mathbb{T}$. Thus, by induction we have

$$[\![P]\!] = t \mapsto \left\{ \begin{array}{l|l} (1,0) & t \in (\mathbb{T} \setminus T_1) \cup (T_2 \setminus T_3) \cup \ldots \\ (0,1) & t \in (T_1 \setminus T_2) \cup (T_3 \setminus T_4) \cup \ldots \end{array} \right.$$

where the sets T_i depend on the button presses (except T_0 which is $\mathbb{T}$),

$$\begin{aligned} T_{2i+1} &= \uparrow \{t \in T_{2i} \mid wb(t) = true\} \\ T_{2i} &= \uparrow \{t \in T_{2i-1} \mid bb(t) = true\} \end{aligned}$$

As we said earlier, the meaning of the overall chess clocks program is the pair obtained by integrating the first and second components of the above value for P. Both these component behaviours are step functions alternating between 1 and 0, so their integrals are straightforward to compute.

5 Related Work

We have discussed our work in relation to Elliott and Hudak's semantics for Fran [EH97] elsewhere in this paper. Ling has identified some problems and suggested extensions to their work [Lin97], but he does not solve the main limitation of their work, namely that it does not account for functions yielding behaviours. Hudak and Wan address the problem of approximation by defining a discrete time semantics for behaviours that corresponds to the implementation, and establishing results that show the convergence of this semantics to an exact continuous time model under suitable conditions [WH00]. Again, this only accounts for behaviours in isolation from functions.

Thompson has suggested a different approach, interpreting Fran programs by translating them into temporal logic formulas [Tho99]. This is an interesting alternative to Elliott and Hudak's denotational approach and to our operational approach, but further work is required to extend this to provide a full semantics for Fran (or for CONTROL). Other formalisms, such as the modal μ-calculus [Koz83], have been used to specify reactive systems, but they differ significantly in approach to CONTROL and generally adopt discrete time. CONTROL is closer to a language than such calculi because there is a straightforward implementation for approximate behaviours [Ell98].

More widely, there many other languages for programming reactive systems. However, most languages adopt a discrete notion of time; for example, Esterel [Ber97], Lustre [HCRP91], Signal [LGLL91] and Imperative Streams [Sch96].

Programs written in these languages are not able to perform operations like integration, which is only valid for continuous time, and can be more difficult to reason about. One continuous-time language is Dannenberg's Arctic [Dan84], which has only an informal description. In considering a formal semantics for Arctic, many of the same issues we have met arise; for example, both languages describe events using time-varying boolean values, so we must define how to react to such events. For this reason we expect that a similar approach to ours could be used to develop a formal semantics for Arctic.

6 Conclusions

We have illustrated CONTROL and its formal semantics with the chess clocks program. Our semantics assigns a meaning to every valid CONTROL program, whereas the semantics Elliott and Hudak gave for Fran [EH97] only interprets the operations on behaviours and events. Moreover, their work describes idealised abstract behaviours, not the ones implemented in the library. Although we also describe idealised behaviours, there is no conflict between the implementation and the semantics because behaviours are built into the language.

Our technique interprets behaviours using transition rules and combines this with the usual operational rules for normal-order functional languages. It is possible to treat many variations of the core language using this technique.

Pragmatically, CONTROL is beyond current techniques for exact real integration and event detection. However, it is possible to implement behaviours using approximation techniques and floating point arithmetic. An approximate implementation would require a different semantics—one that accounts for the errors—but note that this still relies on our idealised semantics otherwise it is not clear what such implementations are approximating.

In summary, we believe that our theory of CONTROL is useful for studying Fran-like languages, both retrospectively to analyse Fran and in future work creating new languages for programming hybrid systems.

Acknowledgements

I would like to thank Conal Elliott for introducing me to this topic during a fascinating internship at Microsoft Research. Mark P. Jones gave superb guidance as my supervisor at Nottingham, and contributed many useful ideas. Simon Thompson and Claus Reinke gave me much useful feedback on this paper. This research was supported by Microsoft Research.

Appendix: Formal semantics

time

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

lift0

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

$*

$$\frac{F \xrightarrow[T_0 \setminus M]{f} F' \quad B \xrightarrow[T_0 \setminus M]{b} B'}{F\ \texttt{\$*}\ B \xrightarrow[T_0 \setminus M]{t \mapsto (f(t))(b(t))} F'\ \texttt{\$*}\ B'}$$

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

$$\frac{B \xrightarrow[T_0 \setminus M]{b} B'}{\texttt{integral}\ B \xrightarrow[T_0 \setminus \emptyset]{\bot_{\mathbb{T} \to \mathbb{R}_\bot}} \varepsilon} \qquad \text{no } \int b \text{ exists}$$

Fig. 3. Transition rules I : Time, lifting, no-change and integral

Formulas for occ, non-occ and bad-cond rules:

$$\begin{aligned} T \quad &= \{t \in T_0 \mid c(t) = true\} \\ Bad &= \{t \in T_0 \mid c(t) = \bot_{\mathbb{B}}\} \end{aligned}$$

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 $$\frac{B \xrightarrow[T_0 \setminus M]{b} B' \quad C \xrightarrow[T_0 \setminus M]{c} C'}{B \ \texttt{until} \ C \ \texttt{then} \ D \xrightarrow[T_0 \setminus M]{b'} \varepsilon} \qquad \uparrow Bad \supseteq M \cup \uparrow T$$

$$b' = t \mapsto \begin{cases} b(t) & t \notin \uparrow Bad \\ \bot & \end{cases}$$

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

Fig. 4. Transition rules II : Reactive behaviours and reduce

References

[Bar84] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[Ber97] G. Berry. The foundations of Esterel, 1997.

[Dan84] R. B. Dannenberg. Arctic: A functional language for real-time control. In *ACM Symposium on LISP and Functional Programming*, pages 96–103, 1984.

[Dan99] Anthony C. Daniels. *A semantics for functions and behaviours.* PhD thesis, The University of Nottingham, 1999. Available from `http://www.cs.ukc.ac.uk/people/staff/acd/thesis.ps`.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *The proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, 1997.

[Ell98] Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*, 1998.

[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.

[HF92] P. Hudak and J. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5):Section T, 1992.

[HJ99] John Hughes and Simon Peyton Jones. Report on the programming language Haskell 98. Available from `http://www.haskell.org/definition/`, 1999.

[Koz83] D. Kozen. Results on the propositional μ-calculus. Technical report, 1983.

[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. Available from `http://pantheon.yale.edu/~ sling/research/690Report.ps.zip`, 1997.

[Mit96] John C. Mitchell. *Foundations for programming languages.* MIT Press, 1996.

[PEL98] John Peterson, Conal Elliott, and Gary Shu Ling. Fran user's manual. `http://www.research.microsoft.com/~ conal/Fran/UsersMan.htm`, 1998.

[Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[Sch96] Enno Scholz. A monad of imperative streams. In *Glasgow FP workshop*, 1996.

[Sco93] Dana S. Scott. A type-theorectical alternative to ISWIM, CUCH, OWHY. *Theorectical computer science*, 121:441–440, 1993.

[Tho99] Simon Thompson. Verifying Fran programs. Available from `http://www.cs.ukc.ac.uk/people/staff/sjt/Fran`, April 1999.

[WH00] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *SIGPLAN '00 Conference on Programming Language Design and Implementation.* ACM Press, June 2000.