CONTROLLING THE BEHAVIOUR OF FUNCTIONAL LANGUAGE SYSTEMS

John Darlington and **Lyndon While**
Department of Computing
Imperial College of Science and Technology
London, SW7
U.K.

ABSTRACT

We present a methodology that allows temporal constraints to be imposed on the behaviour of term-rewriting systems and in particular allows the evaluation order of pure functional programs to be constrained. This permits the use of these languages in applications such as operating systems and real-time control, where control of evaluation order is necessary to achieve a correct implementation.

This control is achieved, in the declarative spirit, by utilising a temporal logic that allows the user to specify, at a high level, the temporal conditions his program must satisfy. The temporal logic is a meta-language which talks about events occurring in the execution of the associated program. The program together with the temporal constraints is then automatically transformed to produce a single program that is guaranteed to behave correctly on any implementation. These techniques are of particular interest when developing programs for parallel asynchronous machines such as ALICE [Cripps et al, 1987] that can exhibit genuinely non-deterministic evaluation (even of deterministic programs).

We detail the temporal specification language, the transformations used and their implementation and give an example showing the use of the methodology with an illustration of its execution on the parallel graph reduction machine ALICE.

1. Functional Languages and Term-rewriting

In this paper we will use the functional language Hope [Burstall et al, 1980]. Some of our examples will be of non-deterministic systems, here we will continue to use Hope syntax but will relax the restriction placed

on Hope (to achieve determinism) that only one equation must match a particular expression being rewritten at any one time.

1.1 Term-rewriting Operational Semantics

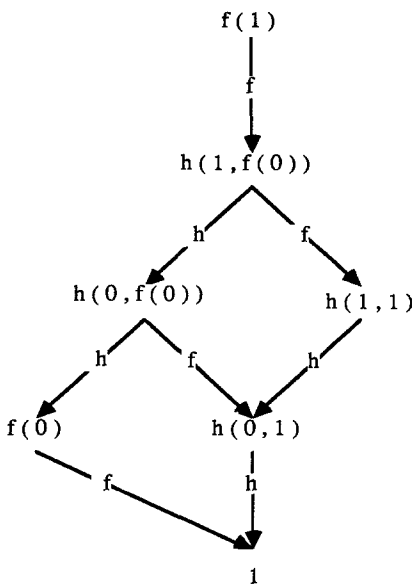
Our view of the execution of a functional program will be that of **term-rewriting**. Thus, for example, given the following program in Hope

```
dec f : num -> num ;
dec h : num # num -> num ;

--- f 0 <= 1 ;
--- f ( n + 1 ) <= h ( n + 1 , f n ) ;

--- h ( 0 , m ) <= m ;
--- h ( n + 1 , m ) <= h ( n , m ) ;
```

the possible ways of using these equations to evaluate a particular expression can be represented by a directed graph whose nodes are expressions and whose arcs are labelled by the equation used to achieve the transition. Thus for f (1) the evaluation graph is



An alternative representation of the possible evaluations is the set of all paths from the (unique) starting node to a result, i.e. any node with no exiting arcs. For the example above we would have

```
{ f;h;h;f , f;h;f;h , f;f;h;h }
```

We identify **events** occurring during the evaluation of a program with the application of a particular equation and our work aims to provide a way of ensuring that these events occur in the required order whenever this is necessary to achieve a correct implementation. This definition of an event will be extended in several ways as we shall see later in the paper.

Sequential implementations traditionally have a fixed rule concerning which particular redex, and therefore equation, to select for the next reduction. Thus **normalising** or **lazy** sequential implementations are based on always choosing the leftmost-outermost redex and have the important property of **safeness**, i.e. they are guaranteed to terminate for evaluations where a normal form exists via some reduction sequence.

However, we are not necessarily confined to always choosing the leftmost redex to guarantee safeness. In particular, on a parallel machine alternative evaluation regimes are practical and desirable. Full parallel reduction, reduce **every** redex, is safe and we certainly want to reduce at least strict arguments concurrently to gain some speed-up from parallelism.

As we can commute reductions, the effect of the parallel evaluation of several redexes is the same as the composition of the individual reductions. Thus parallel reduction has the same effect as a sequential evaluation with variable evaluation order.

The consequence of this is that a programmer using functional languages does not know in what order equations are being used to evaluate his program, nor should we normally want to know. The beauty of functional languages is that the Church-Rosser properties possessed by these languages guarantee that whenever execution terminates the value produced will be the same however it is reached. On a truly parallel machine the order of evaluation may be different from one evaluation of a program to the next.

However, if for example certain of the functions being applied cause **side-effects**, then the perceived **behaviour** of the program may be different from one implementation or particular evaluation to the next. We will argue that there exist implementations where this behavioural component of the program is as important to the correctness of the program as the value returned after execution. We feel that it is contrary to the declarative spirit underlying functional programming for a programmer to achieve control of the program's behaviour by assuming any property of the implementation beyond safeness. Our methodology allows the programmer to specify, using temporal logic, any temporal constraints that need to be imposed on the program's behaviour. These constraints act to remove possible paths from the program's evaluation graph and our transformation process produces a program with exactly this pruned evaluation graph. Thus the transformed program will exhibit the required behaviour under any evaluation strategy. Note also that we will specify the minimum **partial order** that is required between events in contrast to the **total order** often imposed by the other approaches or sequential languages and implementations.

1.2 Non-determinism

Our methodology extends naturally to non-deterministic systems. Where more than one equation is allowed to apply to an expression, different evaluation paths can lead to different results. Many application areas are

naturally non-deterministic (see section six) and are best specified by non-deterministic term-rewriting systems. The style of non-determinism we are addressing is **don't-care** non-determinism. Thus whenever there is a choice of how to reduce a particular redex any matching equation can be chosen and the computation proceeds committed to that choice, with no backtracking or alternative answers. As before with these systems there may be a need to control the system's behaviour. Our method works unchanged, removing incorrect paths from a program's evaluation graph. The final program may, of course, still be non-deterministic even after some constraints have been applied.

2. Synchronisation Methodology

We will present our methodology via a simple example and then expand on the various components. Suppose we had defined "merge" non-deterministically by the following equations:

```
dec merge : list alpha # list alpha -> list alpha ;

--- merge ( nil , x ) <= x ;
--- merge ( x , nil ) <= x ;

M1 : --- merge ( c :: x , y ) <= c :: merge ( x , y ) ;
M2 : --- merge ( x , c :: y ) <= c :: merge ( x , y ) ;
```

Here **M1** and **M2** are labels naming the events, the applications of the particular labelled equations. Only named events can be constrained. Our method ensures that only one constrained event can occur at any one time (we shall refer to the time required to perform the constrained event as the **reduction cycle**). This requirement has the effect that constrained events are sequenced relative to each other but all other events can occur freely and, if possible, in parallel.

We will not address questions of simultaneity between reductions; either we require to impose some ordering on specific pairs of reductions that will necessitate one happening before the other or there is no ordering required between them so normal asynchronous parallel reduction can be used.

Unconstrained, the above program will exhibit a variety of behaviours dependent on the implementation used. Equally there are many ways its behaviour could be constrained. Say, for example, we wished to ensure that merge took items from the two argument lists **alternately**. The first step would be to specify this required behaviour in our temporal language SIAN. An appropriate statement would be

```
M1 → T ( M2 ) , M2 → T ( M1 ) ;
```

which says that if we do **M1** (today) then if we do any constrained equation **T**omorrow, it must be **M2**.

A program's execution is correct relative to a SIAN statement if the statement is true at every point in the execution, i.e. at every rewrite.

The next stage in our methodology is to convert the SIAN expression to a **normal form**. The normal form comprises a list of implications, the consequence of each implication being a conjunction of equation names or negated equation names indicating which equations must **not** be selected depending on the value of the associated condition. The importance of the normal form is that it is defined in such a way that for each implication the condition is testable from what has happened in the past and the consequence is enforceable via the rewrite-rule selection mechanism, thus all of the implications can be kept true. For our example the normal form of the above SIAN expression is

$$\mathbf{Y\ (\ M1\ )} \rightarrow \neg\ \mathbf{M1}\ ,\ \mathbf{Y\ (\ M2\ )} \rightarrow \neg\ \mathbf{M2}\ ;$$

which reads **Y**esterday **M1** implies (today) not **M1** and **Y**esterday **M2** implies (today) not **M2**.

The conditions in the normal form can be evaluated as evaluation progresses by a finite-state machine. The state of the machine is an n-tuple of boolean and integer values that allows the temporal conditions to be evaluated at any time-point (reduction cycle) to determine which rewrite rules are to be disallowed. Transitions in the machine state from one rewrite to the next are derivable systematically from the rewrite rule actually selected. The next step is therefore to construct this finite-state machine. For our example the state is simply a pair of truth-values that record whether **M1** or **M2** respectively were used at the last rewrite (**Y**esterday) and the transition function is simply:

M1

(false , x) => (true , false)

M2

(x , false) => (false , true)

Equations are disallowed by pattern-matching on the values in the state n-tuple; thus in this example, **M1** can only occur if the first parameter is false and **M2** can only occur if the second parameter is false. Notice that if both parameters are false, neither equation is disallowed and so the function is still potentially non-deterministic.

The final stage of our method is to incorporate the finite-state machine into the program being constrained. The state n-tuple is added as an extra argument to the functions being constrained. A pattern is generated for the state for each equation which will automatically disallow the equation (using the normal pattern-matching apparatus) when the state is such that the equation must not be used. The right-hand sides of the equations are transformed so that the state is correctly updated with the information of which equation was actually used for the rewrite.

The end result is an (almost) pure functional program that is totally self-contained and is guaranteed to exhibit the correct temporal behaviour with any implementation. The only non-functional apparatus required is that the data structure representing the state is shared so that updates performed by one rewrite are seen by all the equations controlled by that state. In our example sharing is not necessary as only one redex is present.

The data structure produced for the state is

```
data state == ST ( truval # truval ) ;
```

and the new equations for merge are

```
dec merge : state # list alpha # list alpha -> list alpha ;

--- merge ( s , nil , x ) <= x ;
--- merge ( s , x , nil ) <= x ;

M1 : --- merge ( ST ( false , t ) , c :: x , y )
                <= c :: merge ( new , x , y )
                   where new == ST ( true , false ) ;

M2 : --- merge ( ST ( t , false ) , x , c :: y )
                <= c :: merge ( new , x , y )
                   where new == ST ( false , true ) ;
```

Finally, the starting state for the initial call is determined by evaluating the temporal condition at time zero according to the normal semantics. This gives a call of

```
merge ( init , l1 , l2 )
 where init == ST ( false , false ) ;
```

This allows either equation to be used initially but then evaluation proceeds (deterministically) in an alternating manner.

All of the above processes are fully automatic and a system has been implemented in Hope that takes a Hope program and an associated SIAN expression and automatically produces the suitably enhanced program. The above example and all other examples in this paper have been executed on this system.

3. The Definition of the Temporal Language SIAN

In this section we give full details of the language SIAN used to specify temporal constraints. SIAN is a standard temporal logic augmented with the ability to incorporate predicates over numbers defined in Hope. The BNF of a SIAN expression is as follows.

TemporalExpression :== Implication { , Implication } ;

Implication :== Condition → Condition |
Condition ← Condition |
Condition ↔ Condition |
VariableName = Condition |
(Implication)

Condition :== Event |
VariableName |
Y Condition |
T Condition |
Condition **S** Condition |
Condition **U** Condition |
Condition ∨ Condition |
Condition ∧ Condition |
¬ Condition |
CompFuncn (EventCount { , EventCount }) |
(Condition)

Event :== EquationName

CompFuncn :== alphabetic { alphanumeric }

EventCount :== Event **N** Condition | Numeral

EquationName :== alphabetic { alphanumeric }

VariableName :== alphabetic { alphanumeric }

An occurrence of the name of an equation is interpreted as meaning a use of that equation in a rewrite in the reduction cycle specified by the applied modal operators. An occurrence of the name of a variable is interpreted as meaning the value of the variable in the specified reduction cycle.

The semantics of the four logical operators are such that:

a → b is true if whenever the condition a is true, the condition b is true (read → as "implies").

a ← b is true if whenever the condition b is true, the condition a is true (read ← as "if").

a ↔ b is true if whenever the condition a is true, the condition b is true *and* whenever the condition a is false, the condition b is false (read ↔ as "implies and negates").

v = a is interpreted as meaning that the value of the truth-valued variable v in the present reduction cycle is equivalent to the condition a. The ability to name conditions in this way allows the specification of recursive definitions simply by using the name assigned to the condition in the body of its definition (read = as "is defined as").

The semantics of the four modal operators are such that:

Y (a) is true if the condition a was true in the previous reduction cycle.

T (a) is true if the condition a is true in the next reduction cycle.

a **S** b is true if the condition a was true in every past reduction cycle since, but not necessarily including, the last reduction cycle in which the condition b was true. **S** can therefore be defined in terms of **Y** as:

$$a\ \mathbf{S}\ b \equiv \mathbf{Y}(b) \vee \mathbf{Y}(a) \wedge \mathbf{Y}(a\ \mathbf{S}\ b)$$

a **U** b is true if the condition a is true in every future reduction cycle until, but not necessarily including, the next reduction cycle in which the condition b is true. **U** can therefore be defined in terms of **T** as:

$$a\ \mathbf{U}\ b \equiv \mathbf{T}(b) \vee \mathbf{T}(a) \wedge \mathbf{T}(a\ \mathbf{U}\ b)$$

The boundary conditions for the evaluation of these modal expressions are deduced implicitly, for example in the initial reduction cycle an expression of the form **Y** (e) will evaluate to false, there being no previous cycle in which the event e could have occurred.

The semantics of the **N** operator are such that:

e **N** a denotes the number of times that the event e has occurred since, but not including, the last reduction cycle in which the condition a was true, i.e. it would have the (Hope-like) definition

--- e N a <= if Y (a) **then** 0 **else if Y** (e) **then Y** (e **N** a) + 1 **else Y** (e **N** a) ;

The initial value of such an expression is clearly 0. Values defined in this way can be combined in SIAN using any Hope function (CompFuncn) which operates on integer arguments and which returns a truval. As we will see later, these Hope functions are incorporated unchanged into the final transformed program, so the transformation process does not need to concern itself at all with their semantics or internal form.

The three Boolean operators $\vee$, $\wedge$ and $\neg$ are given their normal interpretations.

Certain (very simple) restrictions are placed on the language to ensure that expressions that are written are sensible and meaningful.

4. Transformation to Normal Form

The BNF for a SIAN expression in normal form is as follows.

TemporalExpression :== Implication { , Implication } ;

Implication :== Condition → Consequence

Consequence :== ¬ Event |
not Event |
Consequence ∧ Consequence |
null

Condition :== Event |
Y Condition |
Condition ∨ Condition |
Condition ∧ Condition |
¬ Condition |
CompFuncn (ParameterRef { , ParameterRef }) |
if (Condition , Condition , Condition) |
ParameterRef

ParameterRef :== <integer index to one of the conditions in the expression>

The definitions of the new operators **if** and **not** are:

if (a , b , c) ≡ if a then 0 else if b then c + 1 else c

a → **not** b ≡ ¬ a → ¬ b

The **if** operator resembles the **N** operator but with the temporal element removed and the recursion made implicit with an accumulating parameter (note that conditions involving the **if** operator are integer-valued, not truth-valued). The value **null** is inserted where a condition has no direct consequence, i.e. for a variable or an internally-generated parameter which has no consequence itself but whose value is needed for the evaluation of another condition (for examples of this, see rules (xx) and (xxxv)).

The significance of the normal form is twofold. Firstly, the condition of each implication refers only to the previous cycle in the computation, thus the value of each condition can be determined purely by reference to the previous values of the conditions and by taking into account the event that occurred in the previous cycle. Secondly, the consequence of each implication is simply a conjunction of event names which must *not* be allowed to occur in the present reduction cycle depending on the value of the associated condition. Thus the normal form can be represented as a finite-state machine and the consequences can be enforced simply by pattern-matching on the values in the state of this machine.

Any legitimate SIAN expression can be converted into the above normal form via the automatic application of the meaning-preserving transformations listed below. The set of rules is confluent and complete; if a case is not included, no transformation is necessary. The notation used is that the a_i are arbitrary conditions, the b_i are arbitrary consequences, the p_i are arbitrary parameters and the e_i, $1 \leq i \leq n$, are the n events needing to be constrained.

4.1 Transformations of Conditions

i. $\mathbf{Y}(\mathbf{T}(a)) \Rightarrow a$

ii. $\mathbf{Y}(a_1 \vee a_2) \Rightarrow \mathbf{Y}(a_1) \vee \mathbf{Y}(a_2)$

iii. $\mathbf{Y}(a_1 \wedge a_2) \Rightarrow \mathbf{Y}(a_1) \wedge \mathbf{Y}(a_2)$

iv. $\mathbf{Y}(\neg a) \Rightarrow \neg \mathbf{Y}(a)$

v. $\mathbf{Y}(a_1 \mathbf{S} a_2) \Rightarrow \mathbf{Y}(a_1) \mathbf{S} \mathbf{Y}(a_2)$

vi. $\mathbf{Y}(a_1 \mathbf{N} a_2) \Rightarrow \mathbf{Y}(a_1) \mathbf{N} \mathbf{Y}(a_2)$

vii. $\mathbf{Y}(f(a_1, \ldots, a_m)) \Rightarrow f(\mathbf{Y}\, a_1, \ldots, \mathbf{Y}\, a_m)$

viii-xiv. Transformations for **T** analogous to the above.

xv. $a_1 \mathbf{S} a_2 \Rightarrow \mathbf{Y}(a_2) \vee \mathbf{Y}(a_1) \wedge \mathbf{Y}(p_i)$

xvi. $a_1 \mathbf{N} a_2 \Rightarrow \mathbf{if}(\mathbf{Y}(a_2), \mathbf{Y}(a_1), \mathbf{Y}(p_i))$

xvii. $\neg(a_1 \vee a_2) \Rightarrow \neg a_1 \wedge \neg a_2$

xviii. $\neg(a_1 \wedge a_2) \Rightarrow \neg a_1 \vee \neg a_2$

xix. $\neg \neg a \Rightarrow a$

The parameter references in rules (xv) and (xvi) are recursive ones to the conditions themselves. The bodies of variable definitions are transformed in the same way as conditions. The definitions are also reversed to form implications, i.e.

xx. $v = a \Rightarrow a \rightarrow \textbf{null}$

Variable references are changed from names to integer parameter references indicating the position of the condition in the expression.

4.2 Transformations of Consequences

xxi. $a \rightarrow e_i \Rightarrow a \rightarrow \neg e_1 \wedge .. \wedge \neg e_{i-1} \wedge \neg e_{i+1} \wedge .. \wedge \neg e_n$

xxii. $a \rightarrow \mathbf{Y}(b) \Rightarrow \mathbf{T}(a) \rightarrow b$

xxiii. $a \rightarrow \neg \mathbf{Y}(b) \Rightarrow \mathbf{T}(a) \rightarrow \neg b$

xxiv. $a \rightarrow \mathbf{T}(b) \Rightarrow \mathbf{Y}(a) \rightarrow b$

xxv. $a \rightarrow \neg \mathbf{T}(b) \Rightarrow \mathbf{Y}(a) \rightarrow \neg b$

xxvi. $a \rightarrow b_1 \mathbf{U} b_2 \Rightarrow \neg b_2 \mathbf{S} a \rightarrow b_1 \vee b_2$

xxvii. $a \rightarrow \neg (b_1 \mathbf{U} b_2) \Rightarrow b_1 \mathbf{S} a \rightarrow \neg b_2$

xxviii. $a \rightarrow b_1 \vee b_2 \Rightarrow a \wedge \neg b_2 \rightarrow b_1 , a \wedge \neg b_1 \rightarrow b_2$

xxix. $a \rightarrow \neg (b_1 \vee b_2) \Rightarrow a \rightarrow \neg b_1 , a \rightarrow \neg b_2$

xxx. $a \rightarrow b_1 \wedge b_2 \Rightarrow a \rightarrow b_1 , a \rightarrow b_2$

xxxi. $a \rightarrow \neg (b_1 \wedge b_2) \Rightarrow a \rightarrow \neg b_1 \vee \neg b_2$

xxxii. $a \rightarrow \neg \neg b \Rightarrow a \rightarrow b$

xxxiii. $b \leftarrow a \Rightarrow a \rightarrow b$

xxxiv. $a \leftrightarrow b \Rightarrow a \rightarrow b , \neg a \rightarrow \neg b$

A little extra complexity is encountered with rule (xxviii) above due to the relative timings of b_1 and b_2. Note also that the implemented system employs a further three transformations for optimisation purposes,

specifically to reduce the number of parameters in the final finite-state machine. Details of these and correctness proofs for all of the transformations can be found in [While, 1987].

Recursive application of these transformation rules will remove all occurrences of **T**, **S**, **N** and **U** from the temporal expression. The final stage in the transformation process is to introduce any auxiliary parameters which are required to allow the temporal expression to be represented as a finite-state machine. The state of such a machine is an n-tuple of Boolean and integer values; its parameters must contain sufficient information about the history of the evaluation to allow the derivation of a set of rewrite-rules for the parameters from one state to the next based *only* on their values in the previous state and the knowledge of which equation was used in the last cycle. This can be achieved only if there are no *nested* modal operators (i.e. **Y**) in any of the conditions; if there are any such occurrences they are removed by introducing a new implication with a null consequence, i.e.

$$\textbf{xxxv.}\ \mathbf{Y}(\mathbf{Y}\,a) \rightarrow b \ \Rightarrow\ \mathbf{Y}\,p_2 \rightarrow b\,,\ \mathbf{Y}\,a \rightarrow \textbf{null}$$

In general, for a condition of this form, one parameter is required to remember the previous value of each of the nested expressions, i.e. there needs to be one parameter for each occurrence of **Y**.

5. Representing the Normal Form Expression as a Finite-state Machine

For each event being constrained in the program we need:

i. A *transition condition* which controls the occurrence of the event according to the state of the finite-state machine.

ii. A *transition rule* which is applied to the parameters of the state when the event does occur and which gives the new values of the parameters after the event.

The transition condition for each event is derived by checking for each parameter if the name of the event occurs in the consequence of the implication indexed by that parameter. If the consequence contains a sub-expression of the form $\neg\, \mathbf{e_i}$, whenever the corresponding condition is *true* the event is not allowed to occur; therefore the parameter is required to be **false**. If the consequence contains a sub-expression of the form **not** $\mathbf{e_i}$, whenever the corresponding condition is *false* the event is not allowed to occur; therefore the parameter is required to be **true**. If the consequence contains no reference to the event, there is no condition on the value of the parameter and its previous value is represented by a new, unique variable name $\mathbf{v_i}$.

To derive the transition rule for the event, each of the parameters of the state is evaluated relative to their previous values derived by the process described above, and taking into account the knowledge that the event itself occurred in the last cycle.

The initial value of the state is obtained by evaluating each of the parameters with a base state where all the values have been set to **false** or **0** (according to their type).

The state information derived can be incorporated into the original program by a few simple steps. Each event corresponds to an equation in the program; the transition condition is added to the pattern of this equation and the transition rule is used as the new value of the state to augment constrained function applications in the body of the equation. Finally, the appropriate data statements and modified equations are constructed using the internal abstract program structures for Hope supported by local Hope-in-Hope implementations.

6. Application Areas Requiring Synchronisation

We identify four application areas where a program's correctness criteria may involve temporal constraints on its behaviour.

i. Reactive Systems: These are systems that interact with the real world via side-effects, such as operating systems and real-time control programs. Arranging for the side-effects to occur in the correct order is part of the specification of such systems (often it is a major component).

ii. Transaction Processing: These are reactive systems where some shared resource is being accessed and modified by several users concurrently. The resource may be shared for reasons of efficiency or because it has some physical manifestation that cannot be duplicated, e.g. seats on a particular aircraft flight. In these areas synchronisation is required to maintain data integrity or consistency of response where several users are interleaving read/write accesses.

iii. Resource Control: The need for finer control over the evaluation order than is currently provided with functional languages can arise for other reasons than the need to program side-effects. For example, we may wish to control the execution order to influence the manner in which the program consumes resources. The text-processing programs discussed in [Hughes, 1983] clearly show that lazy sequential evaluation does not give the programmer fine enough control to be able to achieve optimum performance with regard to time taken and space utilised.

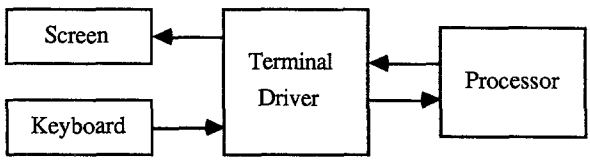
iv. Liveness and Safeness Properties: In non-deterministic systems we may wish to control evaluation order in order to guarantee some desirable property of the overall evaluation, such as the avoidance of **deadlock** or the impossibility of the **starvation** of any resource, that cannot be guaranteed by

relying on purely random rule selection and is very awkward to program explicitly without over-constraining the evaluation.

Our methodology has been successfully applied to a range of examples chosen from all of these areas [While, 1987]. In the next section we detail one such example, a typical reactive system.

7. An Example Application: A Simple Terminal Driver

The problem which we shall examine as an example of our technique is that of specifying a simple driver for an interactive terminal in the following configuration.



The requirements for the driver are:

i. Type-ahead is to be allowed.

ii. Inputs are to be echoed and dealt with in the order in which they are received, and each input is to be echoed before any corresponding output is displayed.

iii. Outputs are to be displayed in the order in which they are produced.

iv. A prompt is to be displayed when the system is quiescent, i.e. when all inputs have been echoed and dealt with and when all outputs have been displayed.

v. The following control interrupts are to be handled:

a. ^C: Purge all queued output and reset the system.
b. ^S: Suspend output until a ^Q is received.
c. ^O: Discard all output produced until the next ^O.

Note that ^S and ^Q do not change the *values* returned; they have a purely temporal effect. ^C and ^O actually cause output to be discarded.

7.1 Specifying the Problem

The driver will be specified as a non-deterministic function where synchronisation is required to ensure its correct behaviour (lazy semantics will be assumed). The following is a list of auxiliary functions which we will assume already exist.

```
dec KeyBoard : void -> list char ;      ! the keyboard

dec OutToScreen : list char -> void ;  ! displays a list of characters on the screen

dec Prompt , Exitchar : char ;          ! the system prompt and the exit character
```

Processing is done in units of a single character; this assumption is not important (and could easily be changed) but it simplifies the equations a little and emphasises the temporal aspects of the problem.

```
dec Process : char -> char ;            ! the processor function
```

The terminal driver function has the declaration:

```
dec TD :  list char          #  list char         ->  list char ;
!         keyboard input     queued output     screen output ;
```

The first argument is the input from the keyboard; each character from this list is placed in the output queue together with the result of processing it. The top level application, which initialises the system, is simply the expression:

```
OutToScreen ( TD ( KeyBoard ( ) , nil ) ) ;
```

Each of the equations of TD expresses one of the functions of the terminal driver.

Characters received from the keyboard are echoed and passed to the processor.

```
inp : --- TD ( c :: l1 , l2 ) <= TD ( l1 , l2 <> [ c , Process c ] ) ;
```

Queued output (echoes and characters from the processor) is sent to the screen.

```
oup : --- TD ( l1 , c :: l2 ) <= c :: TD ( l1 , l2 ) ;
```

^C purges the queued output and resets the system.

```
ControlC : --- TD ( '^C' :: l , _ ) <= "\nInterrupt\n" <> TD ( l , nil ) ;
```

Neither ^S nor ^Q has any functional effect (apart from echoing the control character).

ControlS : --- TD ('^S' :: 11 , 12) <= "\n^S\n" <> TD (11 , 12) ;

ControlQ : --- TD ('^Q' :: 11 , 12) <= "\n^Q\n" <> TD (11 , 12) ;

^O has no direct functional effect; it toggles the system between the normal state and one where any output produced is discarded.

ControlO : --- TD ('^O' :: 11 , 12) <= "\n^O\n" <> TD (11 , 12) ;

When output is produced between two ^O's, it is discarded. This is achieved by a ^Oon event.

ControlOon : --- TD (11 , _ :: 12) <= TD (11 , 12) ;

A prompt is issued simply by placing it in the output stream.

P : --- TD (11 , 12) <= [crlf , Prompt] <> TD (11 , 12) ;

The exit character causes all queued output to be sent to the screen and the driver to terminate.

exit : --- TD (Exitchar :: _ , 1) <= 1 <> [Exitchar] ;

7.2 Constraining the Program to Achieve the Desired Behaviour

The temporal expression needed to constrain the program is the following:

f (inp **N** ControlC , oup **N** ControlC , ControlOon **N** ControlC) $\wedge\neg$ ($\neg$ (inp $\vee$ ControlC) **S** P)
$\leftrightarrow$ P ,
ControlS $\rightarrow$ $\neg$ (oup $\vee$ ControlOon) **U** (ControlQ $\vee$ ControlC) ,
on = **Y** (on) $\wedge$ $\neg$ **Y** (ControlO $\vee$ ControlC $\vee$ P) $\vee$ **Y** ($\neg$ on) $\wedge$ **Y** (ControlO) ,
on $\rightarrow$ $\neg$ oup , $\neg$ on $\rightarrow$ $\neg$ ControlOon ;

where the function f has the definition:

dec f : num # num # num -> truval ;

--- f (x , y , z) <= (2 * x) = (y + z) ;

The first implication tells the system to give a prompt when the system is 'empty' and *not* to give another one until either some input or an interrupt has been received. The second implication represents the rule which bans output between ^S and ^Q. The variable **on** will be true when the ^O toggle is on; in this circumstance any output produced is discarded, otherwise a ^Oon event is disallowed. Note that ^C ends both the ^O toggle and the ^S,^Q toggle.

This expression has been automatically transformed to normal form; the result was:

$$
\begin{aligned}
&\mathbf{Y}\ p_1 \wedge \neg\, \mathbf{Y}\ \text{ControlO} \wedge \neg\, \mathbf{Y}\ \text{ControlC} \wedge \neg\, \mathbf{Y}\ \text{P} \vee \\
&\neg\, \mathbf{Y}\ p_1 \wedge \mathbf{Y}\ \text{ControlO} && \rightarrow \neg\, \text{oup} \wedge \textbf{not}\ \text{ControlOon}\ , \\
&\text{f}\ (\ p_5\ ,\ p_6\ ,\ p_7\ ) \wedge \neg\, p_3 && \rightarrow \neg\, \text{inp} \wedge \neg\, \text{oup} \wedge \neg\, \text{ControlC} \wedge \neg\, \text{ControlO} \wedge \\
& && \quad \neg\, \text{ControlOon} \wedge \neg\, \text{ControlS} \wedge \neg\, \text{ControlQ} \wedge \\
& && \quad \neg\, \text{exit} \wedge \textbf{not}\ \text{P}\ , \\
&\mathbf{Y}\ \text{P} \vee \neg\, \mathbf{Y}\ \text{inp} \wedge \neg\, \mathbf{Y}\ \text{ControlC} \wedge \mathbf{Y}\ p_3 && \rightarrow \textbf{null}\ , \\
&\mathbf{Y}\ \text{ControlS} \vee \neg\, \mathbf{Y}\ \text{ControlQ} \wedge \neg\, \mathbf{Y}\ \text{ControlC} \wedge \mathbf{Y}\ p_4 \\
& && \rightarrow \neg\, \text{oup} \wedge \neg\, \text{ControlOon}\ , \\
&\textbf{if}\ (\ \mathbf{Y}\ \text{ControlC}\ ,\ \mathbf{Y}\ \text{inp}\ ,\ \mathbf{Y}\ p_5\ ) && \rightarrow \textbf{null}\ , \\
&\textbf{if}\ (\ \mathbf{Y}\ \text{ControlC}\ ,\ \mathbf{Y}\ \text{oup}\ ,\ \mathbf{Y}\ p_6\ ) && \rightarrow \textbf{null}\ , \\
&\textbf{if}\ (\ \mathbf{Y}\ \text{ControlC}\ ,\ \mathbf{Y}\ \text{ControlOon}\ ,\ \mathbf{Y}\ p_7\ ) && \rightarrow \textbf{null}\ ;
\end{aligned}
$$

The declaration of the state was:

```
data state == ST ( truval # truval # truval # truval # num # num # num ) ;
```

and the new declaration of TD was:

```
dec TD : state # list char # list char -> list char ;
```

Each equation requires a pattern and rewrite for the state; the derivation of these is also fully automatic. The final program was:

```
inp : --- TD  ( ST ( v1 , false , _ , v4 , v5 , v6 , v7 ) , c :: l1 , l2 )
              <= TD ( new , l1 , l2 <> [ c , Process c ] )
                 where new == ST ( v1 , f ( v5 + 1 , v6 , v7 ) , false , v4 , v5 + 1 , v6 , v7 ) ;

oup : --- TD ( ST ( false , false , v3 , false , v5 , v6 , v7 ) , l1 , c :: l2 )
              <= c :: TD ( new , l1 , l2 )
                 where new == ST ( false , f ( v5 , v6 + 1 , v7 ) and not v3 , v3 , false , v5 ,
                                   v6 + 1 , v7 ) ;

ControlC : --- TD ( ST ( _ , false , _ , _ , _ , _ , _ ) , '^C' :: l , _ )
                  <= "\nInterrupt\n" <> TD ( new , l , nil )
                     where new == ST ( false , f ( 0 , 0 , 0 ) , false , false , 0 , 0 , 0 ) ;

ControlS : --- TD ( ST ( v1 , false , v3 , _ , v5 , v6 , v7 ) , '^S' :: l1 , l2 )
                  <= "\n^S\n" <> TD ( new , l1 , l2 )
                     where new == ST ( v1 , f ( v5 , v6 , v7 ) and not v3 , v3 , true , v5 , v6 ,
                                       v7 ) ;

ControlQ : --- TD ( ST ( v1 , false , v3 , _ , v5 , v6 , v7 ) , '^Q' :: l1 , l2 )
                  <= "\n^Q\n" <> TD ( new , l1 , l2 )
                     where new == ST ( v1 , f ( v5 , v6 , v7 ) and not v3 , v3 , false , v5 ,
                                       v6 , v7 ) ;
```

```
ControlO : --- TD ( ST ( v1 , false , v3 , v4 , v5 , v6 , v7 ) , '^O' :: l1 , l2 )
                <= "\n^O\n" <> TD ( new , l1 , l2 )
                   where new == ST ( not v1 , f ( v5 , v6 , v7 ) and not v3 , v3 , v4 , v5 ,
                                     v6 , v7 );

ControlOon : --- TD ( ST ( true , false , v3 , false , v5 , v6 , v7 ) , l1 , _ :: l2 )
                    <= TD ( new , l1 , l2 )
                       where new == ST ( true, f ( v5 , v6 , v7 + 1 ) and not v3 , v3 , false ,
                                         v5 , v6 , v7 + 1 ) ;

P : --- TD ( ST ( _ , true , _ , v4 , v5 , v6 , v7 ) , l1 , l2 )
          <= [ crlf , Prompt ] <> TD ( new , l1 , l2 )
             where new == ST ( false , false , true , v4 , v5 , v6 , v7 ) ;

exit : --- TD ( ST ( v1 , false , v3 , v4 , v5 , v6 , v7 ) , Exitchar :: _ , 1 )
             <= 1 <> [ Exitchar ]
                where new == ST ( v1 , f ( v5 , v6 , v7 ) and not v3 , v3 , v4 , v5 , v6 , v7 ) ;

--- f ( x , y , z ) <= ( 2 * x ) = ( y + z ) ;
```

with the new top level application

```
OutToScreen ( TD ( init , Keyboard ( ) , nil ) )
 where init == ST ( false , f ( 0 , 0 , 0 ) , false , false , 0 , 0 , 0 ) ;
```

A terminal session demonstrating an execution of this program on the asynchronous parallel machine ALICE can be found in Appendix A.

8. Enhancements to the Method

There are two further important enhancements to the method, the first to increase the expressive power of the temporal language and the second to improve the performance of transformed programs.

8.1 Synchronisation of Individual Function Applications

There are some circumstances in which the identification of an event with the application of a particular rewrite needs to be refined. For example, we may wish to constrain rewrites of a function only in a particular context. Consider the following example.

```
dec apprev , rev : list alpha -> list alpha ;
dec app : list alpha # list alpha -> list alpha ;

F : --- apprev 1 <= app ( 1 , rev 1 ) ;
```

```
--- rev nil <= nil ;
--- rev ( x :: l ) <= app ( rev l , [ x ] ) ;

--- app ( nil , l ) <= l ;
--- app ( x :: l , y ) <= x :: app ( l , y ) ;
```

Simple lazy sequential evaluation is clearly not the optimal strategy to employ when evaluating an application of apprev. The application of app will be completely evaluated before the application of rev is even begun. Indeed, as app will not force the evaluation of its second argument until the whole of its result is required by the calling function, the complete argument list may have to be kept in store for some time, which will be a serious problem if the list is long. What is required is for the evaluation to alternate between applications of app and rev so that they consume the elements of the list at the same rate. Those elements can then be garbage-collected immediately and the store re-used.

This requirement can easily be specified in SIAN but requires extending the language to allow expressions to refer to particular function applications within the bodies of functions and to constrain the choice of those applications in the reduction of the function body. The syntax to represent this is a path expression based on the structure of the function body. An expression for this example is

$$\mathrm{F.} \wedge \neg \mathbf{Y}\ (\ \mathrm{F.arg.2}\ ) \rightarrow \mathbf{T}\ (\ \mathrm{F.arg.2}\ )\ ,\ \mathrm{F.arg.2} \wedge \neg \mathbf{Y}\ (\ \mathrm{F.}\ ) \rightarrow \mathbf{T}\ (\ \mathrm{F.}\ )\ ;$$

which says that if the top-level application of the right-hand side of the equation F (i.e. app) is reduced in this cycle, then the application which is the second argument to the top-level application (i.e. rev) must be reduced in the next cycle, and vice versa. The extra $\neg\ \mathbf{Y}$ conditions are necessary otherwise the constraint would force the entire evaluation of the application of apprev alternately. For each function application constrained in this way, a new function is created with the same definition as the original function and the temporal expression is incorporated into each equation of the new function in the normal way. The normal form for this expression is

$$\mathbf{Y}\ (\ \mathrm{F.}\ ) \wedge \neg\ \mathbf{Y}\ p_3 \rightarrow \neg\ \mathrm{F.}\ ,\ \mathbf{Y}\ (\ \mathrm{F.arg.2}\ ) \wedge \neg\ \mathbf{Y}\ p_4 \rightarrow \neg\ \mathrm{F.arg.2}\ ,$$
$$\mathbf{Y}\ (\ \mathrm{F.arg.2}\ ) \rightarrow \mathbf{null}\ ,\ \mathbf{Y}\ (\ \mathrm{F.}\ ) \rightarrow \mathbf{null}\ ;$$

and the transformed program is:

```
data state == ST ( truval # truval # truval # truval ) ;

dec apprev : list alpha -> list alpha ;
dec app : list alpha # list alpha -> list alpha ;
dec rev' : state # list alpha -> list alpha ;
dec app' : state # list alpha # list alpha -> list alpha ;
```

```
F : --- apprev l <= let s == ST ( false , false , false , false ) in
                        app' ( s , l , rev' ( s , l ) ) ;

--- rev' ( ST ( _ , false , _ , v ) , nil ) <= let s == ST ( false , not v , true , false ) in
                                                  nil ;
--- rev' ( ST ( _ , false , _ , v ) , x :: l ) <= let s == ST ( false , not v , true , false ) in
                                                  app ( rev' ( s , l ) , [ x ] ) ;

--- app' ( ST ( false , _ , v , _ ) , nil , l ) <= let s == ST ( not v , false , false , true ) in
                                                  l ;
--- app' ( ST ( false , _ , v , _ ) , x :: l , y ) <= let s == ST ( not v , false , false , true ) in
                                                  x :: app' ( s , l , y ) ;
```

Notice that the original definitions of app and rev are unchanged for use in other applications (for example the use of app in the second equation of rev). This technique of creating new functions to be identified with events is generally applicable whenever we wish to enhance the definition of an event in the language.

8.2 Dividing up the State

As stated in section two, all constrained events are sequenced relative to each other in the evaluation of a transformed program. This sequencing is required to enforce the behaviour specified by the original temporal expression. In practice, the sequencing effect is brought about by the requirement that the state controlling the program is shared between all of the constrained equations. However, it will often occur that a temporal expression will contain implications which are independent of each other and thus the events constrained by these implications do not actually require to be sequenced. For example, the ^S, ^Q toggle and the ^O toggle in the terminal driver are completely independent of each other (although some of the events which they control are common to them both). In these circumstances, if the state is maintained as a single entity then potential parallelism will be lost and performance may be degraded.

To remedy this situation, before the state data structure is incorporated into the program it is analysed to see if it could possibly be divided up into several smaller states which could act independently during the evaluation. Where this is possible, the model of evaluation outlined in section two (where only one constrained event occurs in each reduction cycle) is generalised so that one constrained event occurs in each reduction cycle *for each state in the program.*

An algorithm has been devised for this analysis and has been incorporated into the Hope transformation system.

9. Related Work

There have been many attempts to apply functional and related languages to problems requiring control of evaluation order ('real-time problems'). We identify three broad schools.

9.1 Temporal Logic

The works closest to our own are those based on the use of temporal logic in some way. [Gabbay, 1986] visualises a problem specification as having two parts, the environment part (E) and the program part (P), together with a temporal control module written in the Since/Until-based temporal logic USF which specifies the desired behaviour of the environment and the program over time. The temporal logic is used as a meta-language for the program P and the paper goes on to discuss the possibility of subsuming P within this meta-language, using the temporal logic as the original imperative langauge for program specification. [Moszkowski, 1985] also develops a language based on temporal logic.

9.2 Stream-processing Functions

Most of the work emanating from direct attempts to apply functional programming to real-time problems have concentrated on the stream-processing approach as exemplified by [Henderson, 1982]. Here the ordering of events in time is achieved by mapping them onto a linear data structure and lazy evaluation is assumed to achieve, for example, the interleaving of input and output. Such approaches usually depart from the pure functional approach by assuming the existence of a non-deterministic merge function. Another problem is that often for realistic-sized problems the intercommunication requirements of the various processes can result in the structure of the program disintegrating. [Stoye, 1986] presents an elegant solution to the latter problem but still requires a departure from the pure functional approach.

The use of *continuation functions* [Karlsson, 1981] that represent the 'rest of the computation' to be performed after a particular function evaluation allows interactive programs to be written in a pure language but at some cost to the structure and readability of the program.

9.3 Communicating Sequential Languages

Starting at the opposite end of the language spectrum with conventional imperative languages there have been many attempts to extend these languages to handle multiple processes and/or processors. Thus CSP [Hoare, 1985] and its descendant occam [May, 1983] give the programmer explicit control over communication and synchronisation and thus can be used to tackle the problems discussed here. The drawback, however, is that they reveal their imperative pedigree by requiring the programmer to be aware of all processes that have to be created and to explicitly control their creation and synchronisation, a complex task for any non-trivial application. This is also true of the more conventional imperative languages such as Ada that have multi-tasking capability.

10. Conclusions

We feel that the methodology presented here allows pure functional languages to be **naturally** applied to important areas where previously they were thought unsuitable. We would criticise many previous attempts to apply functional languages to these areas of either implicitly assuming some property of the implementation used or of requiring the programmer to contort his program so much to achieve the required synchronisation as to destroy the advantages gained from using functional languages in the first place.

Consider, for example, the idea of applying source-to-source transformations to improve a program's behaviour. Such transformations are designed to change a program's **behaviour** while leaving its **meaning** unchanged. If certain properties are required of a program's behaviour but these are achieved implicitly by the way the program is written there is no indication as to what parts of the program can be transformed and what parts must be left untouched to preserve their behaviour. With our method the behavioural requirements are made explicit and the parts of the program to which they apply are clearly delineated so they can be safely mixed with meaning-preserving transformations.

11. Acknowledgements

Particular thanks are due to Dov Gabbay for educating the authors in matters temporal and providing many of the insights that underlie this work. Sebastian Danicic also helped to develop some of the ideas that bore fruit in the work presented here. Finally, we would like to thank all of our colleagues in the Functional Programming Research Group at Imperial College for providing a stimulating and rewarding environment and the U.K. Science and Engineering Research Council for consistently supporting this work.

12. References

1. Burstall, R.M., MacQueen, D.B. and Sanella, D.T., 1980; 'Hope: An Experimental Applicative Language', Proc. 1980 LISP conference, Stanford California. 136-143.
2. Cripps, M.D., Darlington, J., Field, A.J., Harrison, P.G. and Reeve, M.J., 1987; 'The Design and Implementation of ALICE: A Parallel Graph Reduction Machine', Dataflow and Reduction Architectures, ed. S.S. Thakkar, IEEE publications.
3. Gabbay, D., 1986; 'Executable Temporal Logic for Interactive Systems', Internal Report, Department of Computing, Imperial College.
4. Henderson, P., 1982; 'Purely Functional Operating Systems', Functional Programming and its Applications, eds. J. Darlington, P. Henderson and D.A. Turner, Cambridge University Press. 177-189.
5. Hoare, C.A.R., 1985; 'Communicating Sequential Processes', Prentice-Hall International, Englewood Cliffs, New Jersey.
6. Hughes, R.J.M., 1983; 'The Design and Implementation of Programming Languages', Ph.D thesis, Oxford University Computing Laboratory, Programming Research Group. 75-91.

7. Karlsson, K., 1981; 'Nebula: A Functional Operating System', Laboratory for Programming Methodology, Chalmers University of Technology and University of Goteborg, Sweden (draft paper).
8. May, D., 1983; 'Occam', ACM Sigplan Notices, Vol. 18, No. 4. 69-79.
9. Moszkowski, B., 1985; 'Executing Temporal Logic Programs', Internal Report, Computer Laboratory, Cambridge.
10. Stoye, W., 1986; 'Message-based Functional Operating Systems', Science of Computer Programming, Vol. 6, No. 3 (Mag). 291-311.
11. While, R.L., 1987; 'Behavioural Aspects of Term-rewriting Systems', Ph.D thesis, Functional Programming Research Group, Imperial College (in preparation).

Appendix A

This is a terminal log of a session on ALICE running the transformed terminal driver program given in section seven. The prompt is '>', the echoed user input is given in normal text and the processor output is given in underline. In cases where echoed input is discarded by ^C or ^O, the characters input are also given in *italics* at the point at which they are typed. The processor function is defined to change the case of all of the alphabetic characters in the input and to echo non-alphabetic characters. Comments alongside the script are for explanation purposes only and do not form part of the session.

>hHeEllLoO	
>wWoOrRlLdD	: all chars printed
>^S	
^Q	
hHeEllLoO	
wWoOrRlLdD	: output not permitted until ^Q
>^S	: *hello world*
Interrupt	: all of output discarded by ^C (including echoes)
>^O	: *hello world*
^O	
wWoOrRlLdD	: part of output discarded by ^O (amount discarded could vary)
>^O	: *hello world*
Interrupt	: all of output discarded by ^C (including echoes)
>?	