Compiling Real Time Functional Reactive Programming

Dana N. Xu and Siau-Cheng Khoo
Department of Computer Science
School of Computing
National University of Singapore
{xun,khoosc}@comp.nus.edu.sg

ABSTRACT

Most of the past languages for reactive systems are based on synchronous dataflow. Recently, a new reactive language, called Real-Time Functional Reactive Programming (RT-FRP) [18] , has been proposed based on the functional paradigm. The unique feature of this language is the high-level abstraction provided in the form of *behaviors* for continuous-time signals, and *events* for discrete-time signals. RT-FRP also features some performance guarantees in the form of bounded runtime and space usage for each reactive computation step.

In this paper, we propose a new compilation scheme for RT-FRP. Our compilation scheme is based on two key stages. In the first stage, we translate RT-FRP program to an intermediate functional code. This code is deliberately kept at high level for two reasons. First, it is easier for us to validate its correctness. Second, it allows us to apply high-level source-to-source transformation to achieve further optimization. The second stage attempts to compile the intermediate code to a corresponding automata code. Our main novelty is the use of two high-level transformation techniques for this compilation. The first technique, *partial evaluation*, attempts to propagate constant values (wherever feasible) in order to perform more aggressive specialization. The second technique, *tupling*, combines mutually dependent automata together into a composite automaton whenever possible. Both techniques are needed for generating fast target code for RT-FRP.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors – optimization; I.2.2 [Artificial Intelligence]: Automatic Programming – Program transformation.
General Terms
Algorithm, Performance, Languages
Keywords
Reactive System, Partial Evaluation, Tupling

1. INTRODUCTION

Reactive systems are required to react in a timely manner to external events. Their use can be found in a broad range of applications, ranging from high-end consumer products (digital radio, intelligent cookers) to systems used in mission-critical applications (such as air-craft and nuclear-power stations).

Programming these systems poses a great challenge, and a number of programming languages have been proposed over the last two decades. Two main concerns are typically addressed in these languages. Firstly, some features must be available to express signals and events, and how they are transformed by the intended reactive systems. Secondly, we must be able to show that each reaction could be carried out in a safe way using bounded computational resources. Satisfying both these concerns is actually non-trivial as their aims may be contradictory.

We adopt a recently proposed functional reactive programming language for real time reactive system, called RT-FRP, for this purpose. We believe that reactive systems can be conveniently specified using high-level concepts that are available in this language. Reactive system is typically implemented through an automaton. Automata have very simple structure with only states and transitions that can be efficiently implemented. They are also expressive as they can cover a wide range of problems. These properties make them versatile tools for implementing reactive systems [9]. Our main contribution is to show that an efficient implementation exists through our proposal for compiling RT-FRP all the way down to automata code.

The paper is organized as follows. Section 2 gives a brief description of RT-FRP. Section 3 presents some details of compiling RT-FRP to an intermediate functional code; while section 4 highlights some techniques to compile these codes further to automata code. We summarize the paper and propose some future work in section 5.

2. REAL TIME FUNCTIONAL REACTIVE PROGRAMMING (RT-FRP)

In this section, we give an overview of RT-FRP [18] and its possible implementation, as described in [17] that improves upon some of the disadvantages of past reactive languages. Prior to the proposal for RT-FRP, both Elliot and Hudak had invented and developed a functional reactive programming approach, known as FRP [8]. FRP is essentially a declarative programming paradigm that is anchored at two fundamental notions essential to the reactive programming, namely:

ASIA-PEPM'02, September 12-14, 2002, Aizu, Japan.
Copyright 2002 ACM 1-58113-458-4/02/0009 ...$5.00.

a. continuous time-varying behaviors, and

b. discrete event-based reactivity.

Behavior can be abstractly represented as a function of *Time* to value. For example, we may define continuous behavior in Haskell by:

$$type\ Behavior\ a\ =\ Time\ \rightarrow\ a$$

Events on the other hand are essentially time-ordered sequences of discrete values that can be perceived by:

$$type\ Event\ a\ =\ [(Time,\ a)]$$

In RT-FRP, *behavior* and *event* are implemented using sampling technique [9]. Specifically, events can be approximated to instances of behaviors of the *Maybe* types as follows [18]:

$$\begin{array}{l} Event\ a\ \approx\ Behavior\ (Maybe\ a) \\ data\ Maybe\ a\ =\ Nothing\ |\ Just\ a \end{array}$$

When an event occurs, it gives a value *Just x* for some x. If the event does not occur, it provides *Nothing* as a output. Sampling technique makes use of *streams* to capture values of behaviors and events at designated sampling times. Such an approach would use the following implementation for behaviors and events [7].

$$\begin{array}{l} type\ Behavior\ a\ =\ [Time]\ \rightarrow\ [a] \\ type\ Event\ a\ =\ [Time]\ \rightarrow\ [Maybe\ a] \end{array}$$

Here, a behavior is regarded as a function that takes an infinite stream of sample times as input, in order to give an infinite stream of output values. Similarly, an event is also such a function. Using this implementation, time is considered as a primitive behavior that is implemented by

$$\begin{array}{l} time\ ::\ Behavior\ Time \\ time\ =\ \backslash\ ts\ \rightarrow\ ts \end{array}$$

FRP has been used successfully in many reactive programming domains, including animation [16], robotics [15] and graphical user interfaces [6]. However, one of the main weaknesses of FRP is that it does not guarantee resource-boundedness, both in time and in space. This boundedness property is a very crucial aspect of real time reactive systems.

The recently proposed language RT-FRP can be considered as a subset of FRP with a clever combination of syntactic restriction and type system. The imposed restrictions have been shown to be useful for ensuring the resource-boundedness property for space and time that is needed to carry out each reaction [18].

The RT-FRP language consists of two parts: a *reactive* part and a *base language* part. The expressive power of the reactive part is comparable to Synchronous Kahn Network since it allows mutual recursion and higher order functions [3]. Synchronous Kahn Network may have unbounded computation. It has been shown that the cost (in terms of both time and space) of each execution step for a given RT-FRP program is statically bounded, if every expression in the base language is similarly bounded [18]. Figure 1 gives the syntax of both the base language and reactive language of RT-FRP where x and c are syntactic categories of variables and real numbers respectively.

$$\begin{array}{lll}
\multicolumn{3}{l}{\quad \text{— base language syntax}} \\
e & ::= & x \mid c \mid () \mid (e_1,\ e_2) \mid e_\perp \mid\ \perp\ \mid \lambda\, x.e \mid e_1\ e_2 \\
v & ::= & c \mid () \mid (v_1,\ v_2) \mid v_\perp \mid\ \perp\ \mid \lambda\, x.e \\
\\
\multicolumn{3}{l}{\quad \text{— reactive language syntax}} \\
\multicolumn{3}{l}{s,\ ev,\ u\ \in\ SignalR} \\
s,\ ev & ::= & \mathsf{input} \mid \mathsf{time} \mid \mathsf{ext}\ e \mid \mathsf{delay}\ v\ s \mid \\
 & & \mathsf{let\ snapshot}\ x\ \leftarrow\ s_1\ \mathsf{in}\ s_2 \\
 & & s_1\ \mathsf{switch\ on}\ x\ \leftarrow\ ev\ \mathsf{in}\ s_2 \mid \\
 & & \mathsf{let\ continue}\ \{k_j\ x_j\ =\ u_j\}\ \mathsf{in}\ s \mid \\
 & & u \\
u & ::= & s\ \mathsf{until}\ <\ ev_j \Rightarrow\ k_j\ >
\end{array}$$

Figure 1: RT-FRP language syntax.

In Figure 1, e and v represent expressions and values respectively. The terms $e_\perp$ and $\perp$ can be viewed as *Just e* and *Nothing* respectively in Haskell. k is the syntactic category of continuation variables. The construct ext supports the interface between reactive and base languages by exporting snapshots of signal values to the base language. The same construct also allows base language values to be imported back into the signal world. Both snapshots and ext allow us to move between signal world and the value world. They can be implemented using lift operators, as we shall see in the section 3.

3. TRANSLATING RT-FRP TO FUNCTIONAL CODE

In order to realize an efficient implementation for RT-FRP programs, we propose a two-stage compilation that first translates RT-FRP programs to an intermediate functional code, before proceeding to compile the intermediate code to corresponding automata.

In this section, we shall describe the idea for translating RT-FRP to functional code. We choose to use a functional language for intermediate code for two reasons. Firstly, it allows us to verify the correctness of our compilation more easily. Secondly, the use of such a high-level intermediate form also allows us to perform sophisticated optimization. In particular, we are interested in targeting automata (eventually in the form of an imperative program) as our final code.

While the second part of the compilation process has not been fully realized, we have done some preliminary work that showed how high-level program transformation techniques from partial-evaluation [12] and tupling [4] may be adapted to achieve our objective.

3.1 Stream Based Implementation

We propose to translate programs written into RT-FRP into an equivalent functional-style intermediate code (we call it BExp). We adopt stream-based implementation and shall first draw on some operators that have been used in FRP [7] for concise implementation. A stream based implementation is shown in Appendix A.

A reactive behavior can be defined using *untilB*. The operator *untilB* takes a behavior b and an event e, it changes the behavior b when event e occurs. The occurrence of event e generates a new behavior to be followed after the time of its occurrence.

Operator *switcher* assembles a behavior piecewise from an initial one and an event. For example, we can have a signal $s2$ defined as follows

$$s2 \;=\; b1\,`\mathit{switcher}`\,(e1 \;\Longrightarrow\; b2 \,.|.\, e2 \;\Longrightarrow\; b3)$$

which means initial behavior is $b1$, either event $e1$ or $e2$ occurs, it changes. If event $e1$ occurs, it will change to the behavior produced from $(e1 \Longrightarrow b1)$, similarly if event $e2$ occurs, it will change to the behavior produced from $(e2 \Longrightarrow b2)$ where $\Longrightarrow$ is a function defined in Appendix A.

The operator *delay* is not defined in FRP. This operator is meant to delay a given signal by a unit of time. It is actually related to the *timeTransform* operator in FRP. The implementation of *delay* is shown in Appendix A.

It translates expression `ext`, we need to *lift* them so that they can perform computations on lists. Here, a special operator namely (\$*) is used. It takes *Behavior* $(a \rightarrow b)$ and *Behavior* a and returns *Behavior* b. Lifting functions are defined as follows with the help of this operator.

$$\mathit{lift}0 \;=\; \mathit{constantB}$$

where constantB is a function that take a value and produce a list such value, for example, constantB 5 = [5, 5, . . . , 5].

$$\begin{aligned}&\mathit{lift}1\, f\; b1 \;=\; \mathit{lift}0\, f \;\texttt{\$*}\; b1\\ &\mathit{lift}2\, f\; b1\; b2 \;=\; \mathit{lift}1\, f\; b1 \;\texttt{\$*}\; b2\\ &\mathit{lift}3\, f\; b1\; b2\; b3 \;=\; \mathit{lift}2\, f\; b1\; b2 \;\texttt{\$*}\; b3\end{aligned}$$

3.2 Generating Intermediate Functional Code

Translating RT-FRP to functional code requires a special environment : variable environment ε. The variable environment maps snapshot variables to signal variables. The translated signal definitions in functional code are kept in a signal environment κ. This maps signal variables to their corresponding signal definitions. Translation function is denoted by $[\![]\!]_{tr}$, and defined in Figure 2.

Function $\mathit{lookup}_\varepsilon$ takes an expression in the base language and a variable environment and returns a pair of lists of free variables in the expression and their corresponding signal variables. For example, we have $\varepsilon = [\ldots, (a, s1), (b, s2)]$ and $\mathit{lookup}_\varepsilon\ (a + b)$ gives $([a,\ b],\ [s1, s2])$. *newVar* generates new variables. $\sqcup$ updates an variable environment, as well as unions two sets of signal definitions. The correctness of the translation can be easily proven with the extension of the proof in RT-FRP paper.

To illustrate the key idea used in our translation, we shall illustrate how some simple RT-FTP expressions are translated to their functional equivalent:

Given

$$w1 \;=\; \textsf{let snapshot}\ x \leftarrow\ \textsf{time in ext}\ (x\ +\ 1)$$

the following functional intermediate code will be produced:

$$\begin{aligned}w1 &= \mathit{lift}1\,(\backslash\, x \;\rightarrow\; x + 1)\, x_s\\ x_s &= \mathit{time}\end{aligned}$$

Note that ext $(x + 1)$ is translated to a lift operation with the appropriate signals piped into this operation. For any given snapshot x, a corresponding variable x_s is used to capture the signal output at that snapshot. In the above case, x_s is equivalent to time. However, in general, it could be any arbitrary signal expression.

For another example, consider:

$$\begin{aligned}w2 \;=\;& \textsf{let snapshot}\ x \leftarrow\ \textsf{delay}\ 0\ \textsf{ext}\ (x + 1)\ \textsf{in}\\ & \textsf{let snapshot}\ y\ \leftarrow\ \textsf{ext}\ 0\ \textsf{in ext}\ (x + y)\end{aligned}$$

Here two auxiliary signals will be generated, in addition to the main signal as shown below (after renaming).

$$\begin{aligned}w2 \;&=\; = \mathit{lift}2\,(\backslash\, x\ y \;\rightarrow\; x + y)\, x_s\ y_s\\ x_s \;&=\; \mathit{delay}\, 0\,(\mathit{lift}1\,(\backslash\, x \;\rightarrow\; x + 1)\, x_s)\\ y_s \;&=\; \mathit{lift}0\, 0\end{aligned}$$

```
— ————input RT-FRP program—————
w1 = let snapshot x ← time in
     ext (x + 1)
w2 = let snapshot x ← w1 in
     let snapshot y ← time in
     ext (x + y)
w3 = let snapshot t0 ← time in
     let snapshot t1 ← delay 0 time in
     ext (t0 − t1)
w4 = ext (0) switch on x ← w1 in
     ext (x)
w5 = let snapshot y1 ← w1 in
     let snapshot y2 ← w4 in
     let snapshot x ← time in
     let continue {k1 y = ext (y1)
                          until < ext (x) ⇒ k2 >,
                   k2 y = ext (y2)
                          until < ext (x) ⇒ k1 >} in
     ext (y1) until < ext (x) ⇒ k2 >
— —————output in BExp———————
w1 = lift1 (\ x → x + 1) time
w2 = lift2 (\ x → \ y → x + y) w1 time
w3 = lift2 (\ t0 → \ t1 → t0 − t1) time (delay 0 time)
w4 = (lift0 0) switcher w1 ⟹
      (\ x7 → lift1 (\ x → x) x7)
w5 = lift1 (\ y1 → y1) w1 untilB
      lift1 (\ x → x) time ⟹ kj14
kj13 = \ xj15 → lift1 (\ y1 → y1) w1 untilB
        lift1 (\ x → x) time ⟹ kj14
kj14 = \ xj18 → lift1 (\ y2 → y2) w4 untilB
        lift1 (\ x → x) time ⟹ kj13
```

Figure 3: Sample Running

When a time signal is encountered, we replace it by the corresponding time signal. We do a similar translation for the delay operation. Since ext expression is meant for converting a base expression into a signal expression, we translate it to a lift operation. For each snapshot variable relating to a signal w, we first translate w to functional code, assign it a name, and replace the snapshot variable by this name. We show the outcome of our translation with a number of small RT-FRP programs in Figure 3.

4. COMPILING TO AUTOMATA CODE

Reactive system is typically implemented through an automaton. Automata have very simple structure with only states and transitions that can be efficiently implemented. They are also expressive as they can cover a wide range of problems. These properties make them versatile tools for implementing reactive systems. The basic construct behind synchronous languages is the notion of synchronous concurrency. This notion is inspired by Milner's synchronous prod-

$$\llbracket\rrbracket_{tr} :: REqn \rightarrow VEnv \rightarrow KEnv$$

$$\begin{aligned}
&\llbracket \{v_j = er_j\}_{j=1\,\dots\,n}\rrbracket_{tr}\, \varepsilon = \bigsqcup \{\kappa_j\}_{j=1\,\dots\,n} \sqcup \{(xb_1, es_1), \dots, (xb_n, es_n)\}\\
&\quad \textit{where}\ \ (es_j, \kappa_j) = \llbracket\, er_j \,\rrbracket\, \varepsilon' \quad \forall\, j = 1 \dots n\\
&\qquad\qquad \varepsilon' \qquad\quad = \varepsilon \sqcup \{(v_1, xb_1), \dots, (v_n, xb_n)\}\\
&\qquad\qquad xb_j \qquad = newVar \quad \forall\, j = 1 \dots n
\end{aligned}$$

$$\begin{aligned}
&\llbracket \{k_j\ y = er_j\}_{j=1\,\dots\,n}\rrbracket_{tr}\, \varepsilon = (\varepsilon',\, \kappa')\\
&\quad \textit{where}\ \ \kappa' \qquad\quad = \bigsqcup {\kappa_j}_{j=1\,\dots\,n} \sqcup \{(xb_1, \backslash\, y \rightarrow es_1), \dots, (xb_n, \backslash\, y \rightarrow es_n)\}\\
&\qquad\qquad (es_j, \kappa_j) = \llbracket\, er_j \,\rrbracket\, (\varepsilon' \sqcup \{(y, y)\}) \quad \forall\, j = 1 \dots n\\
&\qquad\qquad \varepsilon' \qquad\quad = \varepsilon \sqcup \{(k_1, xb_1), \dots, (k_n, xb_n)\}\\
&\qquad\qquad xb_j \qquad = newVar \quad \forall\, j = 1 \dots n
\end{aligned}$$

$$\llbracket\rrbracket_{tr} :: RExp \rightarrow VEnv \rightarrow (BExp, KEnv)$$

$$\llbracket \mathsf{input} \rrbracket_{tr}\, \varepsilon = (input, \emptyset_\kappa)$$

$$\llbracket \mathsf{time} \rrbracket_{tr}\, \varepsilon = (time, \emptyset_\kappa)$$

$$\begin{aligned}
&\llbracket \mathsf{ext}\ e \rrbracket_{tr}\, \varepsilon = (lift_k\ (\backslash\, v_1 \dots v_k.\ e)\ w_1 \dots w_k, \emptyset_\kappa)\\
&\textit{where}\ ([v_1 \dots v_k],\ [w_1 \dots w_k]) = lookup_\varepsilon\ e\ \varepsilon
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathsf{delay}\ v\ s \rrbracket_{tr}\, \varepsilon = (delay\ v\ s', \kappa)\\
&\textit{where}\ (s', \kappa) = [s]_{tr}\, \varepsilon
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathsf{let\ snapshot}\ x\ s1\ s2 \rrbracket_{tr}\, \varepsilon = (s2', \kappa_1 \sqcup \kappa_2 \sqcup \{(xb_1, s1')\})\\
&\textit{where}\ (s2', \kappa 2) = \llbracket\, s2 \rrbracket_{tr}\, (\varepsilon \sqcup \{(x, xb_1)\})\\
&\qquad\quad (s1', \kappa 1) = \llbracket\, s1 \rrbracket_{tr}\, \varepsilon\\
&\qquad\quad xb_1 \qquad = newVar
\end{aligned}$$

$$\begin{aligned}
&\llbracket\, s1\ \mathsf{switch\ on}\ x\ ev\ s2 \rrbracket_{tr}\, \varepsilon = (s1'\ \text{`}switcher\text{`}\ (ev' \Longrightarrow \backslash\, x \rightarrow s2'),\ \kappa_v \sqcup \kappa_1 \sqcup \kappa_2)\\
&\textit{where}\ (ev', \kappa_v) = \llbracket\, ev \,\rrbracket_{tr}\, \varepsilon\\
&\qquad\quad (s1', \kappa_1) = \llbracket\, s1 \,\rrbracket_{tr}\, \varepsilon\\
&\qquad\quad (s2', \kappa_2) = \llbracket\, s2 \,\rrbracket_{tr}\, (\varepsilon \sqcup \{(x, x)\})
\end{aligned}$$

$$\begin{aligned}
&\llbracket\, s\ \mathsf{until}\ < ev_j \Rightarrow k_j >_{j=1..n} \rrbracket_{tr} \varepsilon =\\
&\qquad\qquad (s'\ \text{`}untilB\text{`}\ (ev'_1 \Longrightarrow k'_1\ .|.\ \dots\ .|.\ ev'_n \Longrightarrow k'_n), \kappa_s \sqcup \bigsqcup \{\kappa_j\}_{j=1\,\dots\,n})\\
&\textit{where}\ (ev'_j, \kappa_j) = \llbracket\, ev_j \rrbracket_{tr}\, \varepsilon\ \ \forall\, j = 1 \dots n\\
&\qquad\quad (s', \kappa_s) = \llbracket\, s \rrbracket_{tr}\, \varepsilon\\
&\qquad\quad k'_j \qquad\ = lookup_\varepsilon\ k_j\ \varepsilon \quad \forall j = 1 \dots n
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathsf{let\ continue}\ \{k_j\ x_j = u_j\}_{j=1..n} in\ s \rrbracket_{tr}\, \varepsilon = (s', \kappa_s \sqcup \kappa')\\
&\textit{where}\ (s', \kappa_s) = \llbracket\, s \,\rrbracket_{tr}\, \varepsilon'\\
&\qquad\quad (\varepsilon', \kappa') = \llbracket\, \{k_j\ x_j = u_j\}_{j=1..n} \,\rrbracket\, \varepsilon\ k'_j = \backslash\, x_j \rightarrow \llbracket\, u_j \rrbracket_{tr} \varepsilon\ \kappa'
\end{aligned}$$

Figure 2: Translation Rules.

uct [13, 14]. In the sampling scheme [9] (RT-FRP belongs to sampling scheme), when a set of automata is composed in parallel, the compound transition for this product (or set) is equivalent to simultaneous transitions for each of the automata. Each automaton considers the output of other automata as its own input for each compound transition. Inter-automata communication is often done through broadcasting.

This motivates us to make an attempt to compile RT-FRP to automata. A brief description of the algorithm is given in Section 4.1. A step-by-step illustration of compiling a specific example is given in Section 4.2. The resulting automata is shown in diagrammatic form in Section 4.3. Section 4.4 shows how we use tupling method [4] to implement synchronous concurrency.

4.1 Description of our Algorithm

As our intermediate code is expressed at high-level, we are able to employ sophisticated transformation techniques to help us derive corresponding automata code. In particular, we shall make use of partial evaluation [12] and unfold/fold techniques [2] to compile a RT-FRP program to automata code.

We first describe the terminology used in our algorithm. Note that streams can be expressed in the form *delay v s*. For any given signal variable x, we define the *instantiate* of x by expressing it as *delay* $x\sharp\ x_{-}$ (we call it *delay* form) where $x\sharp$ and x_{-} are two new variables. $x\sharp$ denotes the current snapshot value, while x_{-} denotes the rest of x (after the current snapshot value). The use of this time-relative annotation can help us avoid the introduction of redundant names. There is another annotation we shall be using (in the next section), namely $x\flat$ which denotes a previous (last most recent) value of x.

We can summarize the various annotations in the following table:

The following is the syntax of our automata code.

$$C := delay\ \ e\ \ C_1 \mid \texttt{case}\, \{e0_i\ :\ delay\ \ e1_i\ \ C_i\}_{i=1,\,\dots,\,m}$$

Each configuration corresponds to a state in an automaton. The first syntactic form represents a direct transition from a state (C) to another state (C_1); e is the output value pro-

Time $(t_i, i = 1, 2, 3, \ldots)$	Notation	Definition
-	x	signal x
t_i	$x\sharp$	value of the signal x at current sample time
t_{i-1}	$x\flat$	value of the signal x at previous sample time
$t_n, n > i$	$x_{\_}$	signal x from $t = i + 1$ onwards

Table 1: Definitions of time-relative annotation

duced during the transition. In the second syntactic form, a state (C) can be transited to several states (C_i), depending on the test $e0_i$ associated with the state C. Again, $e1_i$ is the output value produced when transition to C_i occurs. To ensure the finiteness of configuration state, we allow parameterization of configuration with some values. A *parameterized configuration* is denoted by $C[x_1, \ldots, x_n]$. We shall write C instead of $C[]$ when the configuration has no parameter.

Given a signal expression s, we can *define* a *pre*-configuration $C[x_1, \ldots, x_n] = s'$, such that s' is obtained from s by generalizing all its constants (of type with unbounded variation, such as integers) to variables $x_1, \ldots, x_n$. Note that s' has not yet been converted to the format of automata code. We maintain such pre-configuration in a *pre-configuration* store σ.

Unfolding rules are defined by the operator $[\![]\!]_{uf}$ in Figure 4. Unfolding a signal always yields a signal expressed in *delay* form. Unfolding always terminate – this can be proven by induction on the structure of signal expressions.

Folding a signal expression s requires consultation of the pre-configuration store, σ, for matching against previously defined configuration. It also requires viewing the signal from a particular frame of reference – the time factor. It is defined as follows:

1. Assume the current frame of reference is at time t_{i+1}. Thus, a value produced by s at time t_i is viewed as the previous (most recent) value of s. Consequently, all variables in s of the form $x\sharp$ will be converted to $x\flat$, and $x_{\_}$ to x. The converted signal is denoted by s'.

2. Match s' against configurations in σ. If match succeeds, replace s' by the corresponding configuration name (with appropriate arguments). Return the configuration name and a boolean value $True$. Otherwise, return a null name and a value $False$.

The compilation algorithm for a set of signal definitions in the intermediate functional code, can now be defined as follows:

For each signal definition be of the form $s = es$, *define* a pre-configuration, with name $C[x_1, \ldots, x_n]$. Keep the pre-configuration in the pre-configuration store σ. Furthermore, create an empty configuration store Σ.
While σ is non-empty, repeat the following:

1. Retrieve a pre-configuration $C[x_1, \ldots, x_n] = es1$ from σ.

2. Instantiate $C[x_1, \ldots, x_n]$ to *delay* $c\sharp\ c_{\_}$, with some new variables $c\sharp$ and $c_{\_}$.

$$\begin{aligned}
&[\![\, \mathit{time} \,]\!]_{uf} \;=\; \mathit{delay}\ t\ \mathit{time} \\
&\qquad\qquad\quad \text{— } t \text{ is the current time} \\
&[\![\, \mathit{input} \,]\!]_{uf} \;=\; \mathit{delay}\ i\ \mathit{input} \\
&\qquad\qquad\quad \text{— } i \text{ is the current input} \\
&[\![\, \mathit{delay}\ v\ s \,]\!]_{uf} \;=\; \mathit{delay}\ v\ s
\end{aligned}$$

$$\begin{aligned}
&[\![\, s1\ \mathit{switcher}\ ev \implies f]\!]_{uf} \\
&= \mathsf{let} \mathit{delay}\ v\ s1' \;=\; [\![\, s1]\!]_{uf} \\
&\qquad \mathit{delay}\ w\ ev' \;=\; [\![\, ev]\!]_{uf} \\
&\quad \mathsf{in\ case}\ \{\ w \;=\; \mathit{Nothing}\ : \\
&\qquad\qquad \mathit{delay}\ v\ (s1'\ \mathit{switcher}\ ev' \implies f), \\
&\qquad\quad w \;=\; (\mathit{Just}\ e) : \\
&\qquad\qquad \mathit{delay}\ v\ ((f\ e)\ \mathit{switcher}\ ev' \implies f)\}
\end{aligned}$$

$$\begin{aligned}
&[\![\, s1\ \mathit{untilB}\ (ev_1' \implies k_1' \ \ldots \\
&\qquad\qquad .|.\ ev_j' \implies k_j' \ \ldots \\
&\qquad\qquad .|.\ ev_n' \implies k_n')]\!]_{uf} \\
&= \mathsf{let}\ \mathit{delay}\ v\ s1' \;=\; [\![\, s1]\!]_{uf} \\
&\qquad \mathit{delay}\ w_1\ ev_1' \;=\; [\![\, ev_1]\!]_{uf} \\
&\qquad\quad \vdots \\
&\qquad \mathit{delay}\ w_j\ ev_j' \;=\; [\![\, ev_j]\!]_{uf} \\
&\qquad\quad \vdots \\
&\qquad \mathit{delay}\ w_j\ ev_n' \;=\; [\![\, ev_n]\!]_{uf} \\
&\quad \mathsf{in\ case}\ \{(w_1,\ \ldots,\ w_n) \;==\; (\mathit{Nothing},\ \ldots,\ \mathit{Nothing})\ : \\
&\qquad\qquad \mathit{delay}\ v\ (s1'\ \mathit{untilB}\ (ev_1' \implies k_1' \ \ldots \\
&\qquad\qquad\qquad\qquad\qquad .|.\ ev_j' \implies k_j' \ \ldots \\
&\qquad\qquad\qquad\qquad\qquad .|.\ ev_n' \implies k_n')), \\
&\qquad\qquad (\ldots,\ w_{j-1},\ w_j,\ w_{j+1},\ \ldots) \;== \\
&\qquad\qquad\quad (\ldots,\ \mathit{Nothing},\ \mathit{Just}\ e,\ \mathit{Nothing},\ \ldots)\ : \\
&\qquad\qquad\qquad\qquad\qquad \mathit{delay}\ v\ (k_j'\ e)\}
\end{aligned}$$

$$[\![\, \mathit{lift}0\ e \,]\!]_{uf} \;=\; \mathit{delay}\ e\ (\mathit{lift}0\ e)$$

$$\begin{aligned}
&[\![\, \mathit{lift}1\ (\backslash\ x1 \rightarrow\ e)\ (\mathit{delay}\ v1\ v)]\!]_{uf} \\
&=\ \mathit{delay}\ e[v1/x1]\ (\mathit{lift}1\ (\backslash\ x1 \rightarrow \backslash\ x2 \rightarrow\ e)\ v)
\end{aligned}$$

$$\begin{aligned}
&[\![\, \mathit{lift}2\ (\backslash\ x1 \rightarrow \backslash\ x2 \rightarrow\ e)\ (\mathit{delay}\ v1\ v)\ (\mathit{delay}\ w1\ w)\,]\!]_{uf} \\
&=\ \mathit{delay}\ e[v1/x1, w1/x2]\ (\mathit{lift}2\ (\backslash\ x1 \rightarrow \backslash\ x2 \rightarrow\ e)\ v\ w)
\end{aligned}$$

$$\vdots$$
$$\vdots$$

Figure 4: Unfolding rules

3. Derive the configuration for $C[x_1, \ldots, x_n]$ by subjecting $es1$ to the following steps:

   (a) $es1_a \leftarrow$ Instantiate all signal variables in $es1$;

   (b) $es1_b \leftarrow [\![es1_a]\!]_{uf}$. The $es1_b$ will be of the form

   $$\mathtt{case}\ \{t_i\ :\ \mathit{delay}\ v_i\ s_i\}_{i=1,\ldots,m}.$$

   (c) $es1_c \leftarrow \mathit{case}\ \{t_j' \rightarrow \mathit{delay}\ v_j'\ s_j\}_{i=1,\ldots,n}$, where t_i', and v_i' are the result of simplifying the expressions t_i and v_i respectively.

   (d) For each branch *delay* $v_i'\ s_i$ in $es1_c$, unify it with *delay* $c\sharp\ c_{\_}$ created at step 2, resulting in a signal of the form *delay* $v_i''\ s_i''$. Update $es1_c$ to yield $es1_d$.

   (e) For s_i'' in each branch *delay* $v_i''\ s_i''$ of $es1_d$, *fold* it against the configuration store σ. If folding fails, *define* a pre-configuration for s_i'', and replace s_i'' in $es1_d$ by the pre-configuration name. Add the new pre-configuration definition in σ. Applying *fold* and *define* to all branches of $es1_d$ yields a new expression, denoted by $es2$.

(f) Add $C[x_1, \ldots, x_n] = es2$ to the final configuration store Σ, and delete the pre-configuration $C[x_1, \ldots, x_n]$ from σ.

During unfold/folding/define process, compiled code is obtained which corresponds to some automata. Thus, the number of states generated increases with the depth of unfolding.

Termination of the algorithm

Proof. (Sketch) As every unfolding step always terminates, we just need to show that there are finite number of configurations. This is guaranteed when there are finite number of different signal expressions $es1_d$ produced at Step (d) of the algorithm (and thus finite number of $es2$.)

Since $es1_d$ is of the form $case\ \{e0'_i \rightarrow delay\ e1''_i\ s''_i\}_{i=1,\ldots,n}$, we need to show that there are only finite number of different $e0_i$, $e1''_i$ and s''_i.

Both $e0_i$ and $e1''_i$ are of finite variation because – thanks to the abstracting process performed during *define* step – all constants of unbounded variety have been generalized, and all time-related variables used referred to at most three time stamp: previous (most-recent), current, and future.

The proof for s''_i having finite variation can proceed by an induction over the structure of the expressions produced by *unfold* process, together with the fact that all values used in s''_i are of finite variation – an argument similar to the proof for $e0_i$ and $e1''_i$ to have finite variation. □

4.2 An Example to Illustrate the Compilation from RT-FRP to Automata

In this section, we shall illustrate an example adapted from [10] with three mutual-recursive signals, namely z, x, n; and an input signal, called *input* (see Figure 5). We shall show how these signals are compiled into three mutual-dependent automata. During the compilation process, every finitely bounded values are instantiated, where possible, in order to provide more specialization.

```
z = let snapshot a ← delay True x in
    let snapshot b ← delay True z in
    let snapshot c ← input in
    ext (if a then b else ((a and b) or c))
x = let snapshot a ← delay True x in
    let snapshot b ← z in
    ext (if a then False else b)
n = let snapshot a ← delay True x in
    let snapshot b ← delay 0 n in
    ext (if a then 0 else b + 1)
```

Figure 5: A program written in RT-FRP

Our first step is to generate an intermediate functional code from the RT-FRP program. In the case of above example, we can obtain the following:

Compile RT-FRP to functional code (in pretty printed form)

$$\begin{aligned} x &= \mathit{lift2}\,(\backslash\ a \rightarrow \backslash\ b \rightarrow \mathit{if}\ a\ \mathit{thenFalse}\ \mathit{else}\ b)\\ &\quad (\mathit{delay}\ \mathit{True}\ x)\ z\\ z &= \mathit{lift3}\,(\backslash\ a \rightarrow \backslash\ b \rightarrow \backslash\ c \rightarrow\\ &\quad \mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ ((a\ \&\&\ b)\ \|\ c))\\ &\quad (\mathit{delay}\ \mathit{True}\ x)\ (\mathit{delay}\ \mathit{True}\ z)\ \mathit{input}\\ n &= \mathit{lift2}\,(\backslash\ a \rightarrow \backslash\ b \rightarrow \mathit{if}\ a\ \mathit{then}\ 0\mathit{else}\ b+1)\\ &\quad (\mathit{delay}\ \mathit{True}\ x)\ (\mathit{delay}\ 0\ n)\end{aligned}$$

During our transformation, we shall be defining new configurations that correspond to states in the automata code at each intermediate step. At each configuration, we shall leave signals and constants that have bounded values unchanged, while all infinitely bounded expressions (e.g. integers) are appropriately generalized. This strategy provides aggressive optimization without causing non-terminating transformation. In the case of signal, n, a suitably generalized configuration is shown below.

Generalize Integer constants (In this case, change the integers 0 at then-branch, 1 at else-branch and 0 in the delay to $c1$, $c2$, $c3$ respectively).

$$\begin{aligned} &\mathit{nconfig}_1[c1, c2, c3] = n\\ &\mathit{where}\ n = \mathit{lift2}\,(\backslash\ a \rightarrow \backslash\ b \rightarrow \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b+c2)\\ &\qquad (\mathit{delay}\ \mathit{True}\ x)\,(\mathit{delay}\ c3\ n)\end{aligned}$$

We are now in a position to perform partial evaluation of our configuration. There are two signals in the code. In order to obtain an appropriate result, we instantiate the two signals in the following step.

Instantiate signals and substitute

$$\begin{aligned} n &= \mathit{delay}\ n\sharp\ n_{\_} \quad \text{— [1]}\\ x &= \mathit{delay}\ x\sharp\ x_{\_}\\ n &= \mathit{lift2}\,(\backslash\ a \rightarrow \backslash\ b \rightarrow \mathit{if}\ a\ \mathit{then}\ c1\mathit{else}\ b+c2)\\ &\quad (\mathit{delay}\ \mathit{True}\ (\mathit{delay}\ x\sharp\ x_{\_}))\,(\mathit{delay}\ c3\ (\mathit{delay}\ n\sharp\ n_{\_}))\end{aligned}$$

Proceeding further with our specialization, we shall perform an unfold step for the lift operation, as shown below.

Unfold once and partial evaluate v

$$\begin{aligned} n &= \mathit{delay}\ (\mathit{if}\ \mathit{True}\ \mathit{then}\ c1\ \mathit{else}\ c3+c2)\\ &\quad (\mathit{lift2}(\backslash\ a \rightarrow \backslash\ b \rightarrow \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b+c2)\\ &\quad (\mathit{delay}\ x\sharp\ x_{\_})(\mathit{delay}\ n\sharp\ n_{\_}))\\ &= \mathit{delay}\ c1\ (\mathit{lift2}\,(\backslash\ a \rightarrow \backslash\ b \rightarrow\\ &\quad \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b+c2)\\ &\quad (\mathit{delay}\ x\sharp\ x_{\_})\,(\mathit{delay}\ n\sharp\ n_{\_})) \quad \text{— [2]}\end{aligned}$$

As both [1] and [2] are identical, we can unify the two equations in order to obtain the following two equations.

Unification

$$\begin{aligned} &\text{— Unify [1] and [2], we have}\\ n\sharp &= c1\\ n_{\_} &= \mathit{lift2}\,(\backslash\ a \rightarrow \backslash\ b \rightarrow \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b+c2)\\ &\quad (\mathit{delay}\ x\sharp\ x_{\_})\,(\mathit{delay}\ c1\ n_{\_})\end{aligned}$$

The first result is substituted back into our new configuration, while the second equation is used as a new configuration.

Shifting the frame of reference for $n_{\_}$ to t_{i+1}

$$\begin{aligned} &\mathit{lift2}(\backslash\ a \rightarrow \backslash\ b \rightarrow \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b+c2)\\ &(\mathit{delay}\ x\flat\ x)\,(\mathit{delay}\ c1\ n)\end{aligned}$$

since the above signal does not match the definition of $nconfig_1$, folding fails. Thus, we introduce a new configuration in the next step.

Introduce $nconfig_2$ for $n_{\_}$

$$\begin{aligned}
&\mathit{nconfig}_1[c1, c2, c3] \ = \ \mathit{delay}\ c1\ \mathit{nconfig}_2[c1, c2, c1]\\
&\mathit{nconfig}_2[c1, c2, c3] \ = \ n\\
&\mathit{where}\ n \ = \ \mathit{lift2}\ (\backslash\ a \ \rightarrow \ \backslash\ b \ \rightarrow \ \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b + c2)\\
&\qquad\qquad (\mathit{delay}\ x\flat\ x)\ (\mathit{delay}\ c3\ n)
\end{aligned}$$

We add the final definition of $nconfig_1$ to Σ, and definition of $nconfig_2$ to σ. Since σ is non-empty, we repeat the compilation loop again.

We retrieve $nconfig_2$ from σ and instantiate it followed by unfolding. This is applied as shown below.

Instantiation: $x \ = \ \mathit{delay}\ x\sharp\ x_{\_}$
$\qquad\qquad n \ = \ \mathit{delay}\ n\sharp\ n_{\_}$

$$\begin{aligned}
n \ = \ &\mathit{lift2}\ (\backslash\ a \ \rightarrow \ \backslash\ b \ \rightarrow \ \mathit{if}\ a\ \mathit{then}\ c1 \mathit{else}\ b + c2)\\
&(\mathit{delay}\ x\flat\ (\mathit{delay}\ x\sharp\ x_{\_}))\ (\mathit{delay}\ c3\ (\mathit{delay}\ n\sharp n_{\_}))\\
= \ &\mathit{delay}\ (\mathit{if}\ x\flat\ \mathit{then}\ c1\ \mathit{else}\ b + c2)\\
&\quad (\mathit{lift2}\ (\backslash\ a \rightarrow \backslash\ b \rightarrow \ \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b + c2)\\
&\qquad (\mathit{delay}\ x\sharp\ x_{\_})\ (\mathit{delay}\ (c3 + c2)\ n_{\_}))
\end{aligned}$$

One of the snapshot, $x\flat$, happens to be a boolean value. In order to aggressively partially evaluate this configuration, we dispatch this value based on two possible values, True and False, resulting in:

case $x\flat \ == \ \mathit{False}$:

$$\begin{aligned}
n \ = \ &\mathit{delay}\ (c3 + c2)\ (\mathit{lift2}\ (\backslash\ a \rightarrow \backslash\ b \rightarrow\\
&\quad \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b + c2)\\
&\quad (\mathit{delay}\ x\sharp\ x_{\_})\ (\mathit{delay}\ n\sharp\ n_{\_}))
\end{aligned}$$

— after unification, we have $n\sharp = c3 + c2$

$$\begin{aligned}
n \ = \ &\mathit{delay}\ (c3 + c2)\ (\mathit{lift2}\ (\backslash\ a \ \rightarrow \ \backslash\ b \ \rightarrow\\
&\quad \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b + c2)\\
&\quad (\mathit{delay}\ x\sharp\ x_{\_})\ (\mathit{delay}\ (c3 + c2)\ n_{\_}))
\end{aligned}$$

— fold with $\mathit{nconfig}_2[c1, c2, c3 + c2]$

$$n \ = \mathit{delay}\ (c3 + c2)\ \mathit{nconfig}_2[c1, c2, c3 + c2]$$

case $x\flat \ == \ \mathit{True}$:

$$\begin{aligned}
n \ = \ &\mathit{delay}\ c1\ (\mathit{lift2}\ (\backslash\ a \ \rightarrow \ \backslash\ b \ \rightarrow\\
&\quad \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b + c2)\\
&\quad (\mathit{delay}\ x\sharp\ x_{\_})\ (\mathit{delay}\ n\sharp\ n_{\_}))
\end{aligned}$$

— after unification, we have $n\sharp = c1$

$$\begin{aligned}
n \ = \ &\mathit{delay}\ c1\ (\mathit{lift2}\ (\backslash\ a \ \rightarrow \ \backslash\ b \ \rightarrow\\
&\quad \mathit{if}\ a\ \mathit{then}\ c1\ \mathit{else}\ b + c2)\\
&\quad (\mathit{delay}\ x\sharp\ x_{\_})\ (\mathit{delay}\ c1\ n_{\_}))
\end{aligned}$$

— fold with $nconfig_2[c1, c2, c1]$

$$n \ = \ \mathit{delay}\ c1\ \mathit{nconfig}_2[c1, c2, c1]$$

In both branches, the subsequent configuration encountered matches with a previous configuration, allowing our specialization to terminate successfully.

The final automata code is shown below.

Final program for n:

$$\begin{aligned}
&\mathit{nconfig}_1[c1, c2, c3] \ =\\
&\quad \mathit{delay}\ c1\ \mathit{nconfig}_2[c1, c2, c1]\\
&\mathit{nconfig}_2[c1, c2, c3] \ =\\
&\quad \mathit{case}\ \{x\flat \ == \ \mathit{False} \ :\\
&\qquad\qquad n = \mathit{delay}\ (c3 + c2)\\
&\qquad\qquad\qquad \mathit{nconfig}_2[c1, c2, c3 + c2],\\
&\qquad\quad x\flat \ == \ \mathit{True} \ :\\
&\qquad\qquad n = \ \mathit{delay}\ c1\ \mathit{nconfig}_2[c1, c2, c1]\}
\end{aligned}$$

By a similar sequence of transformation, we are also able to generate the following automata codes for both x and z.

Using the algorithm, we can get final programs for x and z as follows.

Final Program for x:

$$\begin{aligned}
&\mathit{xconfig}_1 \ = \ \mathit{delay}\ \mathit{False}\ \mathit{xconfig}_2\\
&\mathit{xconfig}_2 \ = \ \mathit{delay}\ z\sharp\ \mathit{xconfig}_3\\
&\mathit{xconfig}_3 \ =\\
&\quad \mathit{case}\ \{z\flat \ == \ \mathit{False} \ :\\
&\qquad\quad x \ = \ \mathit{delay}\ z\sharp\ \mathit{xconfig}_3,\\
&\qquad z\flat \ == \ \mathit{True} \ :\\
&\qquad\quad x \ = \ \mathit{delay}\ \mathit{False}\ \mathit{xconfig}_2\}
\end{aligned}$$

Final program for z:

$$\begin{aligned}
&\mathit{zconfig}_1 \ = \ \mathit{delay}\ \mathit{True}\ \mathit{zconfig}_2\\
&\mathit{zconfig}_2 \ =\\
&\quad \mathit{case}\ \{x\flat \ == \ \mathit{True} \ :\\
&\qquad\quad z \ = \ \mathit{delay}\ \mathit{True}\ \mathit{zconfig}_2,\\
&\qquad x\flat \ == \ \mathit{False} \ :\\
&\qquad\quad z \ = \ \mathit{delay}\ \mathit{True}\ \mathit{zconfig}_3\}\\
&\mathit{zconfig}_3 \ =\\
&\quad \mathit{case}\ \{x\flat \ == \ \mathit{True} :\\
&\qquad\quad z \ = \ \mathit{delay}\ z\flat\ \mathit{zconfig}_3,\\
&\qquad x\flat \ == \ \mathit{False} \ :\\
&\qquad\quad z \ = \ \mathit{delay}\ \mathit{input}\sharp\ \mathit{zconfig}_3\}
\end{aligned}$$

4.3 Automata Diagram

We can illustrate diagrammatically the automata code generated with a node for each state. A transition from a node A to another node B may depend on a test, indicated by a ? symbol. The output produced during at each transition is indicated by a ! symbol. A state may also be parameterized (as shown in figure 6) by some values. Parameters of a state can be updated by the transition leading to that state. Figure 6, 7 and 8 shows the automata generated for the signals n, x and z respectively.

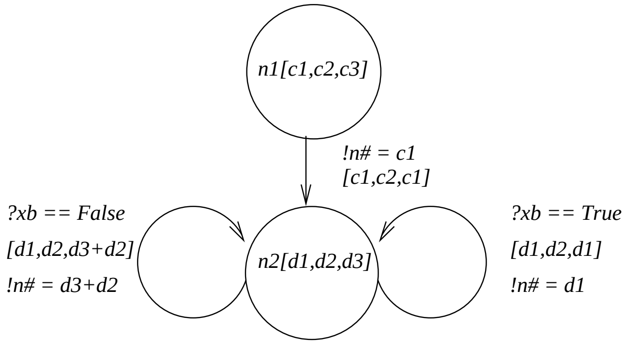

Figure 6: Automaton generated for the signal n.

4.4 Tupled Automata

The generated automata code may actually be interdependent. Instead of allowing them as separate processes, we can in fact simplify the control by merging the mutually-dependent signals together. The transformation process to accomplish this is known as *tupling*. Here, we collect a set of configurations together, and begin to partially evaluate the components simultaneously. Where successful, the result of tupling transformation generates a tupled (or composite) automata, with each transition representing a synchronous step by the mutually-dependent signals. One of the key advantages of tupling is that output from one signal, can immediately be made available (or visible) to its dependent signal(s). This can facilitate more specialization.

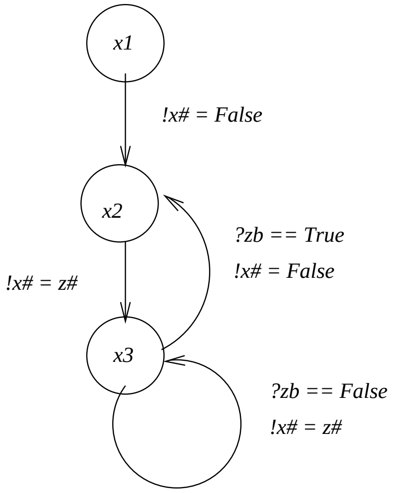

Figure 7: Automaton generated for the signal x.

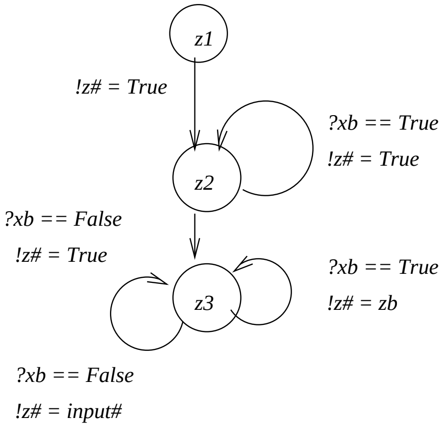

Figure 8: Automaton generated for the signal z.

We define a *tupled configuration* (abbreviated as t-config) as follows.

$$\begin{aligned}&M[v_{11},\ \ldots,\ v_{mn}]\ :=\\&\quad(C_1[v_{11},\ \ldots,\ v_{1n}],\ \ldots,\ C_m[v_{m1},\ \ldots,\ v_{mn}])\end{aligned}$$

The following *tupling algorithm* takes in a set of automata in their automata-code format, and returns a tupled automata in the same format. The result is stored in Ω.

1. Define a t-config by grouping the starting nodes of all existing automata. Keep the new configuration in a store ω.

2. While ω is non-empty, do the following steps:

   (a) Retrieve a t-config from ω. Let it be
   $$M := (C_1, \ldots, C_n).$$
   We shall omit the parameters in our presentation. We shall call C_i a source configuration.

   (b) Unfold each source configuration C_i once to get
   $$\begin{aligned}M\ :=\ (&\mathit{case}\ \{t_{1j}\ :\ \mathit{delay}\ e_{1j}\ C_{1j}\ \},\ \ldots,\\&\mathit{case}\ \{t_{nj}\ :\ \mathit{delay}\ e_{nj}\ C_{nj}\ \})\end{aligned}$$

   (c) Form all possible tuples by combining a branch from each component configuration.
   $$\begin{aligned}M\ :=\ \mathit{case}\ \{\ &t_{1j_k}\ \&\&\ \ldots\ \&\&\ t_{nj_l}\ :\\&\mathit{delay}\ (e_{1j_k}, \ldots, e_{nj_l})\ (C_{1j_k}, \ldots, C_{nj_l})\}\end{aligned}$$

   (d) Simplify the test in each branch of M above. Delete those branches whose test simplified to *False*.

   (e) For each t-config occurring in each branch of M, Fold it against those in ω. If that fails, define it as a new t-config and add the new definition in ω. Consequently, M will once again become a piece of automata code. Add it to the output store Ω and delete it from the store ω.

This tupling algorithm terminates because there are finite number of states in each of the composite automata.

Let us illustrate this technique using our earlier example from the section 4.2. There were three initial configurations from the three signals, n, z and r. These configurations are grouped together to form the following tupled configuration.

$$\begin{aligned}&\mathit{mconfig}_{111}[c1, c2, c3]\ =\ (n, x, z)\\&\mathit{where}\ (n, x, z)\ =\ (\mathit{nconfig}_1[c1, c2, c3], \mathit{xconfig}_1, \mathit{zconfig}_1)\end{aligned}$$

We unfold all the component configuration by following their respective transitions, resulting:

$$\begin{aligned}&\mathit{delay}\ (c1,\ \mathit{False},\ \mathit{True})\\&\quad(\mathit{nconfig}_2[c1, c2, c1], \mathit{xconfig}_2, \mathit{zconfig}_2)\end{aligned}$$

The new tuple of components can now be defined as a new configuration $\mathit{mconfig}_{222}$.

$$=\ \mathit{delay}\ (c1,\ \mathit{False},\ \mathit{True})\ \mathit{mconfig}_{222}[c1, c2, c1]$$

Following shows the steps involved in performing tupling algorithm:

$$\begin{aligned}&\mathit{mconfig}_{222}[c1, c2, c3]\ =\ (n, x, z)\quad \text{— } c1, False, True\\&\mathit{where}\\&(n, x, z)\ =\ (\mathit{nconfig}_2[c1, c2, c3], \mathit{xconfig}_2, \mathit{zconfig}_2)\\&\mathit{case}\ x\flat\ ==\ \mathit{False}\quad\text{— yes}\\&(n, x, z)\ =\ (\mathit{nconfig}_2[c1, c2, c3], \mathit{xconfig}_2, \mathit{zconfig}_2)\\&\qquad\quad =\ \mathit{delay}\ (c3 + c2, z\sharp, \mathit{True})\\&\qquad\qquad(\mathit{nconfig}_2[c1, c2, c3 + c2], \mathit{xconfig}_3, \mathit{zconfig}_3)\\&\qquad\quad\ \text{— since } x\sharp = z\sharp\\&\qquad\quad =\ \mathit{delay}\ (c3 + c2, \mathit{True}, \mathit{True})\\&\qquad\qquad(\mathit{nconfig}_2[c1, c2, c3 + c2], \mathit{xconfig}_3, \mathit{zconfig}_3)\\&\qquad\quad\ \text{— introduce } \mathit{mconfig}_{233}\\&\qquad\quad =\ \mathit{delay}\ (c3 + c2, \mathit{True}, \mathit{True})\\&\qquad\qquad\mathit{mconfig}_{233}[c1, c2, c3 + c2]\end{aligned}$$

$$\begin{aligned}&\mathit{mconfig}_{233}[c1, c2, c3]\ =\ (n, x, z)\quad\text{— } c3 + c2, True, True\\&\mathit{where}\\&(n, x, z)\ =\ (\mathit{nconfig}_2[c1, c2, c3], \mathit{xconfig}_3, \mathit{zconfig}_3)\\&\mathit{case}\ x\flat\ ==\ \mathit{True},\ z\flat\ =\ \mathit{True}\quad\text{— yes}\\&(n, x, z)\ =\ (\mathit{nconfig}_2[c1, c2, c3], \mathit{xconfig}_3, \mathit{zconfig}_3)\\&\qquad\quad =\ \mathit{delay}\ (c1, \mathit{False}, \mathit{True})\\&\qquad\qquad(\mathit{nconfig}_2[c1, c2, c1], \mathit{xconfig}_2, \mathit{zconfig}_3)\\&\qquad\quad\ \text{— introduce } \mathit{mconfig}_{223} \text{ – (c1,False,True)}\\&\qquad\quad =\ \mathit{delay}\ (c1, \mathit{False}, \mathit{True})\ \mathit{mconfig}_{223}[c1, c2, c1]\end{aligned}$$

$$\begin{aligned}&\mathit{mconfig}_{233a}[c1, c2, c3]\ =\ (n, x, z)\\&\qquad\qquad\qquad\qquad\text{— } (c3 + c1, \mathit{input}\sharp, \mathit{input}\sharp)\\&\mathit{where}\\&(n, x, z)\ =\ (\mathit{nconfig}_2[c1, c2, c3], \mathit{xconfig}_3, \mathit{zconfig}_3)\\&\mathit{case}\ x\flat\ ==\ \mathit{False},\ z\flat\ ==\ \mathit{False}\end{aligned}$$

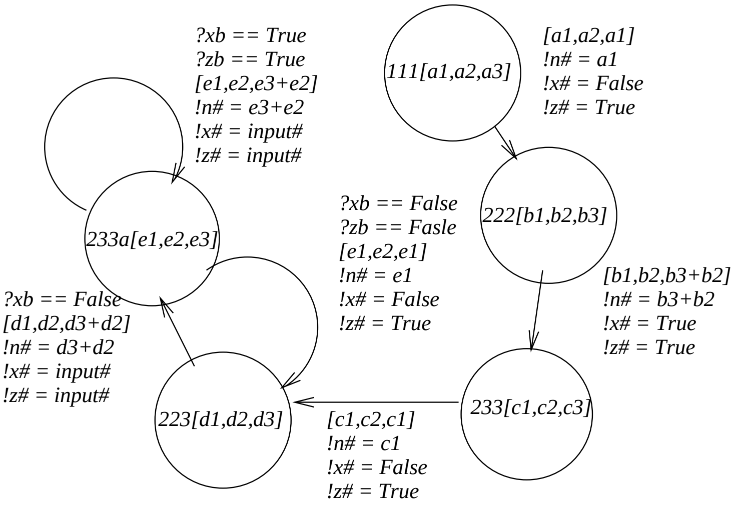

Figure 9: Tupled Automaton.

$$\begin{aligned}
(n,x,z) &= (\mathit{nconfig}_2[c1,c2,c3], \mathit{xconfig}_3, \mathit{zconfig}_3)\\
&= \mathit{delay}\ (c3+c2, z\sharp, \mathit{input}\sharp)\\
&\quad (\mathit{nconfig}_2[c1,c2,c3+c2], \mathit{xconfig}_3, \mathit{zconfig}_3)\\
&\ \text{— folded with } mconfig_{233}\\
&\ \text{— } (c3+c2, \mathit{input}\sharp, \mathit{input}\sharp)\\
&= \mathit{delay}\ (c3+c2, z\sharp, \mathit{input}\sharp)\\
&\quad \mathit{mconfig}_{233a}[c1,c2,c3+c2]
\end{aligned}$$

$\mathit{case}\ x\flat\ ==\ \mathit{True},\ z\flat\ ==\ \mathit{True}$ — yes

$$\begin{aligned}
(n,x,z) &= (\mathit{nconfig}_2[c1,c2,c3], \mathit{xconfig}_3, \mathit{zconfig}_3)\\
&= \mathit{delay}\ (c1, \mathit{False}, z\flat)\\
&\quad (\mathit{nconfig}_2[c1,c2,c1], \mathit{xconfig}_2, \mathit{zconfig}_3)\\
&\ \text{— introduce } mconfig_{223} \quad \text{— (c1,False,True)}\\
&= \mathit{delay}\ (c1, \mathit{False}, \mathit{True})\ \mathit{mconfig}_{223}[c1,c2,c1]
\end{aligned}$$

$\mathit{mconfig}_{223}[c1,c2,c3] = (n,x,z)$ — (c1,False,True)
where

$$\begin{aligned}
(n,x,z) &= (\mathit{nconfig}_2[c1,c2,c3], \mathit{xconfig}_2, \mathit{zconfig}_3)
\end{aligned}$$

$\mathit{case}\ x\flat\ ==\ \mathit{False}$ — yes

$$\begin{aligned}
(n,x,z) &= (\mathit{nconfig}_2[c1,c2,c3], \mathit{xconfig}_2, \mathit{zconfig}_3)\\
&= \mathit{delay}\ (c3+c2, z\sharp, \mathit{input}\sharp)\\
&\quad (\mathit{nconfig}_2[c1,c2,c3+c2], \mathit{xconfig}_3, \mathit{zconfig}_3)\\
&\ \text{— folded with } mconfig_{233a}\\
&\ \text{— } (c3+c1, \mathit{input}\sharp, \mathit{input}\sharp)\\
&= \mathit{delay}\ (c3+c2, \mathit{input}\sharp, \mathit{input}\sharp)\\
&\quad \mathit{mconfig}_{233a}[c1,c2,c3+c2]
\end{aligned}$$

Through the above tupling transformation, we can obtain a 5-state tupled automata, which computes three output values simultaneously with each transition. Some branches which are dead code were eliminated in the process. The diagrammatical representation is shown in figure 9.

$$\begin{aligned}
&\mathit{mconfig}_{111}[c1,c2,c3] =\\
&\quad \mathit{delay}\ (c1, \mathit{False}, \mathit{True})\ \mathit{mconfig}_{222}[c1,c2,c1]
\end{aligned}$$

$$\begin{aligned}
&\mathit{mconfig}_{222}[c1,c2,c3] =\\
&\mathit{delay}\ (c3+c2,\ \mathit{True},\ \mathit{True})\ \mathit{mconfig}_{233}[c1,c2,c3+c2]
\end{aligned}$$

$$\begin{aligned}
&\mathit{mconfig}_{233}[c1,c2,c3] =\\
&\mathit{delay}\ (c1, \mathit{False}, \mathit{True})\ \mathit{mconfig}_{223}[c1,c2,c1]
\end{aligned}$$

$$\begin{aligned}
&\mathit{mconfig}_{223}[c1,c2,c3] =\\
&\mathit{delay}\ (c3+c2,\ \mathit{input}\sharp,\ \mathit{input}\sharp)\ \mathit{mconfig}_{233a}[c1,c2,c3+c2]
\end{aligned}$$

$$\begin{aligned}
&\mathit{mconfig}_{233a}[c1,c2,c3] =\\
&\mathit{case}\ \{x\flat\ =\ \mathit{False\ and}\ z\flat\ =\ \mathit{False}\ :\\
&\qquad \mathit{delay}\ (c3+c2,\ \mathit{input}\sharp,\ \mathit{input}\sharp),\\
&\qquad \mathit{mconfig}_{233a}[c1,c2,c3+c2]\\
&\qquad x\flat\ =\ \mathit{True\ and}\ z\flat\ =\ \mathit{True}\ :\\
&\qquad \mathit{delay}\ (c1, \mathit{False}, \mathit{True})\ \mathit{mconfig}_{223}[c1,c2,c1]\}
\end{aligned}$$

5. CONCLUSIONS AND FUTURE WORKS

We have introduced a high-level and yet systematic approach to building reactive systems based on RT-FRP. This approach is centered towards building signals that are either continuous or event-driven. We note that a small number of language constructs is sufficient for supporting a wide range of possible programs. Both recursive equations and mutual-recursive continuations are expressible in RT-FRP.

The main focus of this project is to develop a high-level framework for compiling reactive language, such as RT-FRP. We first tanslate RT-FRP programs to an intermediate functional code. Use of this functional code facilitates high-level compilation to automata. Our framework is unique in that it utilizes two high-level transformation techniques, namely, partial evaluation and tupling, and it provides a systmatic methodology for compilation.

Partial evaluation is used to specialize each signal to its corresponding automata. We adapt the technique to the context of stream-based semantics by including shifting frame of references during folding. Furthermore, through specializing variables whose values are of finite variation (such as boolean values), we are able to eliminate many dynamic tests in our code. This optimization is important as it can help improve the run-time performance of reactive systems. Better performance is obtained at the expense of larger program size, since more states may be generated during partial evaluation.

Each of the signals would normally be executed independently. However, a number of these signals may be mutually dependent on each other. To facilitate passing of values among signals, as well as to simplify global control, we employ tupling technique by combining related signals into a composite signal.

That high-level transformation techniques can be systematically applied to obtain compiled automata code is the distinctive contribution of this work. This is in contrast with the compilation of Lustre, in which specific compilation techniques, such as “data driven” and “demand driven” control synthesis, were introduced [10]. While the automata code produced by our framework are not as compact as those with specific techniques, we believe that the goal for high-level transformation techniques to drive compilation still remains promising.

Wan *et. al.* presentated, in [19], a set of compilation rules for *Event-Driven FRP* (E-FRP), a dialect of FRP. Despite syntactic similarity between E-FRP and RT-FRP, the former obeys interrupt-driven semantics, instead of stream-based. Wan *et. al.* compiled E-FRP programs to an imperative language (called SimpleC), and described specific techniques for optimizing the generated imperative code. It would be interesting to investigate the issues related to compiling E-FRP in our framework.

On the RT-FRP aspect, we believe that the resulting (tupled) automaton adheres to the safety properties of its original RT-FRP program; ie., resource and time boundedness. We have yet to formally prove this, though.

Looking into the future, we find it useful to re-design the language to make it more event-oriented. Many applications in reactive systems are largely event-oriented, and the provision of a deterministic and concurrent language can be the basis of a powerful development paradigm.

Secondly, we are interested in exploring how sized analysis [11, 5], can be used to extend our language, while still preserving the space and time boundedness property.

Lastly, apart from compiling to automata, we would like to explore the possibility of compiling directly to hardware. This ultimate compilation route allows concurrency to be exploited to the fullest and is becoming more feasible with wider adoption of FPGA technology. Presently, a variant of Haskell, known as Lava [1], allows compilation to VHDL and FPGA. We hope to adopt Lava as a target for our hardware compilation strategy.

6. ACKNOWLEDGMENT

We would like to thank Wei-Ngan Chin and everyone residing in the Programming Language Laboratory of National University of Singapore for their valuable feedback through numerous discussions. We also would like to thank the anonymous referees for their valuable comments. This work has been supported by the research grants R-252-000-092-112.

7. REFERENCES

[1] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. In *ACM Intl. Conference on Functional Programming*, pages 176–187, Baltimore, Maryland, USA, June 1998. ACM Press.

[2] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.

[3] P. Caspi and M. Pouzet. Synchronous kahn network. In *ACM Intl. Conference on Functional Programming*, pages 176–187, Philadephia, May 1996. ACM Press.

[4] W.N. Chin. Towards an automated tupling tactic. In *3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993.

[5] W.N. Chin and S.C. Khoo. Calculating sized types. In *2000 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72, Boston, Massachusetts, United States, January 2000.

[6] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, Italy, 2001.

[7] Conal Elliott. Functional implementations of continuous modeled animation. In *PLILPALP*, 1998.

[8] Conal Elliott and Paul Hudak. Functional reactive system. In *ACM Intl. Conference on Functional Programming*, Philadephia, June 1997. ACM SIGPLAN.

[9] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *10th International Conference on Computer-Aided Verification*, CAV'98 Vancouver (B.C.), LNCS 1427, Springer Verlag, June 1998.

[10] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[11] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM Principles of Programming Languages Conference*, pages 410–423. ACM Press, January 1996.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. *International Series in Computer Science*, (0-13-020249-5 (pbk)):44–67, June 1993.

[13] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Department, Edimburgh University, 1981.

[14] R. Milner. Calculi for synchrony and asynchrony. *International Series in Computer Science*, (25(3)), July 1983.

[15] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Practical Aspects of Declarative Languages*, Florence, September 1999.

[16] Simon Thompson. A functional reactive animation of a lift using fran. Technical Report 5-98, Computing Laboratory, University of Kent, May 1998.

[17] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Vancouver, BC, Canada, 2000.

[18] Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *ACM Intl. Conference on Functional Programming*, Florence, September 2001.

[19] Z. Wan, W. Taha, and P. Hudak. Event-driven frp. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, Portland, Oregon, USA, January 2002.

APPENDIX

A. STREAM BASED IMPLEMENTATION

The following definitions are largely taken from [7]. The symbol u represents a stream of user input synchronizes with time.

```
type Behavior a = [Time] → [a]
type Event a = [Time] → [Maybe a]

time :: Behavior Time
time = \ ts → ts

input :: Event User
input = \ ts → u

delay :: a → Behavior a → Behavior a
delay v s = \ ts → v : (s ts)

⟹ :: Event a → (a → b) → Event b
e1 ‘⟹‘ f = \ ts → loop ts (e1 ts)
where loop (:ts′) (Nothing : es)
          = Nothing : (loop ts′ es)
         loop (:ts′) (Just a : es)
          = (Just (f a)) : (loop ts′ es)


untilB :: Behavior a →
               Event (Behavior a) → Behavior a
b ‘untilB‘ e = \ ts → loop ts (b ts) (e ts)
where loop (:ts′) (x : xs′) (Nothing : mb)
          = x : (loop ts′ xs′ mb)
         loop ts (x : xs′)(Just bn : _)
          = x : (bn ts)


switcher :: Behavior a →
                 Event (Behavior a) → Behavior a
s ‘switcher‘ e = \ ts → loop ts (s ts) (e ts)
where loop (:ts′) (x : xs′)(Nothing : mb)
          = x : (loop ts′ xs′ mb)
         loop (:ts′) (x : xs′) (Just bn : mb)
          = x : (loop ts′ (bn ts′) mb)


.|. :: Event a → Event a → Event a
e1 ‘.|.‘ e2 = \ ts → zipWith aux (e1 ts) (e2 ts)
where aux Nothing Nothing = Nothing
         aux (Just x) _ = Just x
         aux _ (Just x) = Just x


fb ‘$∗‘ xb = \ ts → zipWith ($) (fb ts) (xb ts))
lift0 = constantB
lift1 f b1 = lift0 f $∗ b1
lift2 f b1 b2 = lift1 f b1 $∗ b2
lift3 f b1 b2 b3 = lift2 f b1 b2 $∗ b3
 :
 :
```