Event-Driven FRP*

Zhanyong Wan, Walid Taha, and Paul Hudak

Department of Computer Science,
Yale University, New Haven, CT, USA
{wan-zhanyong,taha,hudak-paul}@cs.yale.edu

Abstract. Functional Reactive Programming (FRP) is a high-level declarative language for programming reactive systems. Previous work on FRP has demonstrated its utility in a wide range of application domains, including animation, graphical user interfaces, and robotics. FRP has an elegant continuous-time denotational semantics. However, it guarantees no bounds on execution time or space, thus making it unsuitable for many embedded real-time applications. To alleviate this problem, we recently developed *Real-Time FRP* (RT-FRP), whose operational semantics permits us to formally guarantee bounds on both execution time and space. In this paper we present a formally verifiable compilation strategy from a new language based on RT-FRP into imperative code. The new language, called *Event-Driven FRP* (E-FRP), is more tuned to the paradigm of having multiple external events. While it is smaller than RT-FRP, it features a key construct that allows us to compile the language into efficient code. We have used this language and its compiler to generate code for a small robot controller that runs on a PIC16C66 micro-controller. Because the formal specification of compilation was crafted more for clarity and for technical convenience, we describe an implementation that produces more efficient code.

1 Introduction

Functional Reactive Programming (FRP) [12, 19] is a high-level declarative language for programming reactive systems. A *reactive system* is one that continually responds to external stimuli. FRP has been used successfully in domains such as interactive computer animation [6], graphical user interface design [5], computer vision [15], robotics [14], and control systems. FRP is sufficiently high-level that, for many domains, FRP programs closely resemble equations naturally written by domain experts to specify the problems. For example, in the domain of control systems, our experience has been that we can go from a traditional mathematical specification of a controller to running FRP code in a matter of minutes.

FRP is implemented as an embedded language in Haskell [7], and so provides no guarantees on execution time or space: programs can diverge, consume large

* Funded by NSF CCR9900957, NSF ITR-0113569, and DARPA F33615-99-C-3013 and DABT63-00-1-0002

amounts of space, and introduce unpredictable delays in responding to external stimuli. In many applications FRP programs have been "fast enough," but for real-time applications, careful FRP programming and informal reasoning is the best that one can do.

In trying to expand the scope of FRP to these more demanding settings, we recently identified a subset of FRP (called RT-FRP) where it can be statically guaranteed that programs require only limited resources [20]. In RT-FRP there is one global clock, and the state of the whole program is updated when this clock ticks. Hence every part of the program is executed at the same frequency.

One key application domain that we are interested in is micro-controllers. In addition to being resource bounded, these hardware devices have a natural programming paradigm that is not typical in functional programming languages. In particular, they are event-driven. In such systems, an event affects only a statically decidable part of the system state, and hence there is no need to propagate the event throughout the whole system. Unfortunately, RT-FRP was not designed with this concern in mind.

This paper addresses the issue of compiling a variant of RT-FRP, called *Event-driven FRP (E-FRP)*. In this variant, the RT-FRP global clock is generalized to a set of events. This framework makes it clear that there is no need for RT-FRP's built-in notion of time, since we can now implement it as a special case of an event stream that is treated as a clock. Indeed, now we can have many different time bases, not necessarily linearly related, and in general can deal with complex event-driven (and periodic) reactive systems in a natural and effective manner. We have found that E-FRP is well-suited for programming interrupt-driven micro-controllers, which are normally programmed either in assembly language or some dialect of C.

In what follows, we present the basic ideas of E-FRP with an example from our work on the RoboCup challenge [16].

1.1 A Simple Robot Controller in E-FRP

The Yale RoboCup team consists of five radio-controlled robots, an overhead vision system for tracking the two teams and a soccer ball, and a central FRP controller for guiding the team. Everything is in one closed control loop. Even in this relatively small system, certain components must be developed in low level languages. Of particular interest to us is the controller running on-board the robots. Based on the speed sensor feedback and increase/decrease speed instructions coming from the central FRP controller, the on-board controller must determine the exact signals to be sent to the motors. In each robot, a PIC16C66 micro-controller [13] is used to execute the code that controls the robot. Initially, this low-level controller was programmed in a specialized subset of C, for which a commercial compiler targeting PIC16C66 exists. Written at this level, however, these controllers can be quite fragile, and some important features of the design of the controller can easily become obscured. Reasoning about the combination of the main FRP controller and these separate controllers

is also non-trivial. This problem would not exist if the controllers are written in FRP, or a subset thereof.

The Physical Model Each robot has two wheels mounted on a common axis and each wheel is driven by an independent motor. For simplicity we focus on one wheel, and do not discuss the interaction between the two.

The amount of power sent to the motor is controlled using a standard electrical engineering technique called *Pulse-Width Modulation* (PWM): instead of varying the voltage, the power is rapidly switched off and on. The percentage of time when the power is on (called the *duty cycle*) determines the overall power transmission.

Each wheel is monitored through a simple stripe detection mechanism: The more frequently the stripes are observed by an infrared sensor, the faster the wheel is deemed to be moving. For simplicity, we assume the wheel can only go in one direction.

The Computational Model The main goal of the controller is to regulate the power being sent to the motor, so as to maintain a desired speed ds for the wheel. The control system is driven by five independent interrupt sources:

- *IncSpd* and *DecSpd* increment and decrement the desired speed;
- *Stripe* occurs 32 times for every full revolution of the wheel;
- $Timer0$ and $Timer1$ are two timers that occur at regular, but different, intervals. The frequency of $Timer0$ is much higher than that of $Timer1$.

A register output determines the ON/OFF state of the motor.

The E-FRP Execution Model Before presenting the code for the controller, some basic features of the E-FRP execution model need to be explained.

As with FRP, the two most important concepts in E-FRP are *events* and *behaviors*. Events are time-ordered sequences of discrete event occurrences. Behaviors are values that react to events, and can be viewed as time-indexed signals. Unlike FRP behaviors that can change over continuous time, E-FRP behaviors can change value only when an event occurs.

E-FRP events are mutually exclusive, meaning that no two events ever occur simultaneously. This decision avoids potentially complex interactions between event handlers, and thus greatly simplifies the semantics and the compiler.

Finally, on the occurrence of an event, execution of an E-FRP program proceeds in *two distinct phases*: The first phase involves carrying out computations that depend on the previous state of the computation, and the second phase involves updating the state of the computation. To allow maximum expressiveness, E-FRP allows the programmer to insert annotations to indicate in which of these two phases a particular change in behavior should take place.

Source Code

```
ds = init x = 0 in
     { IncSpd ⇒ x + 1,
       DecSpd ⇒ x − 1},
s = init x = 0 in
     { Timer1 ⇒ 0 later,
       Stripe ⇒ x + 1},
dc = init x = 0 in
     { Timer1 ⇒
         if x < 100 and s < ds
              then x + 1
         else if x > 0 and s > ds
              then x − 1
         else x},
count = init x = 0 in
     { Timer0 ⇒
         if x ≥ 100 then 0
         else x + 1},
output = if count < dc then 1 else 0
```

Target Code

```
(IncSpd,
     ⟨ds := ds⁺ + 1;
      output := if count < dc then 1 else 0⟩,
     ⟨ds⁺ := ds;
      output := if count < dc then 1 else 0⟩),
(DecSpd,
     ⟨ds := ds⁺ − 1;
      output := if count < dc then 1 else 0⟩,
     ⟨ds⁺ := ds;
      output := if count < dc then 1 else 0⟩),
(Timer1,
     ⟨s⁺ := 0;
      dc := if dc⁺ < 100 and s < ds
              then dc⁺ + 1
            else if dc⁺ > 0 and s > ds
              then dc⁺ − 1
            else dc⁺;
      output := if count < dc then 1 else 0⟩,
     ⟨s := s⁺; dc⁺ := dc;
      output := if count < dc then 1 else 0⟩),
(Stripe,
     ⟨s := s⁺ + 1;
      output := if count < dc then 1 else 0⟩,
     ⟨s⁺ := s;
      output := if count < dc then 1 else 0⟩),
(Timer0,
     ⟨count := if count⁺ >= 100 then 0
                 else count⁺ + 1;
      output := if count < dc then 1 else 0⟩,
     ⟨count⁺ := count;
      output := if count < dc then 1 else 0⟩)
```

Fig. 1. The Simple RoboCup Controller (SRC)

An E-FRP Controller Figure 1 presents an E-FRP controller, together with the target code produced by our compiler. In what follows we explain the definition of each of *ds*, *s*, *dc*, *count*, and *output* in detail.

The desired speed *ds* is defined as an "init ... in ..." construct, which can be viewed as a state machine that changes state whenever an event occurs. The value of *ds* is initially 0, then it is incremented (or decremented) when *IncSpd* (or *DecSpd*) occurs. The local variable x is used to capture the value of *ds* just before the event occurs.

The current speed measurement s is initially 0, and is incremented whenever a stripe is detected (the *Stripe* interrupt). This value, however, is only "inspected"

when $Timer1$ occurs, at which point it is reset to zero. The later annotation means that this resetting is not carried out until all other first-phase activities triggered by $Timer1$ have been carried out (that is, it is done in the second phase of execution).

The duty cycle *dc* directly determines the amount of power that will be sent to the motor. On $Timer1$, the current speed *s* is compared with the desired speed *ds* (Recall that *s* is reset by $Timer1$ in the second phase, after all the first-phase computations that depend on the penultimate value of *s* have been completed). If the wheel is too slow (or too fast) then the duty cycle is incremented (or decremented). Additional conditions ensure that this value remains between 0 and 99.

The current position in the duty cycle is maintained by the value *count*. This value is incremented every time $Timer0$ occurs, and is reset every 100 $Timer0$ interrupts. Hence it counts from 0 to 99 repeatedly. The actual 0/1 signal going to the motor is determined by the value *output*. Since *output* is 1 only when $count < dc$, the larger *dc* is, the more power is sent to the motor, and hence the more speed. Note that *output* is updated whenever *count* or *dc* changes value.

Our compiler produces SimpleC, a restricted form of C. The target code is a group of event handlers. Each event handler is responsible for one particular event source, and consists of two sequences of assignments, one for each of the two phases. When the event occurs, the two assignment sequences are executed in turn to update the values of the behaviors.

Highlights The source program is easy to read and concise. It is also a minor extension on a subset of FRP, and hence inherits its denotational semantics. As we will show in this paper, it also has a simple operational semantics from which it is immediate that the program is resource bounded. The compiler generates resource bounded, but bloated, target code. However, we are able to substantially enhance the target code by a four-stage optimization process.

1.2 Contribution and Organization of this Paper

The key contribution of this paper is identifying a subset of FRP that can be compiled into clearly efficient and resource bounded code, and that is suitable for programming event-driven applications. The compilation strategy and the optimization techniques we present are also interesting.

Section 2 presents the syntax and semantics of E-FRP. Section 3 introduces the target language. Section 4 defines a compilation strategy from E-FRP into imperative code. Section 5 explains the optimization techniques.

2 Syntax and Semantics of E-FRP

Notation 1 We write $\langle f_j \rangle^{j \in \{1..n\}}$ for the sequence $\langle f_1, f_2, \ldots, f_n \rangle$. We omit the superscript $j \in \{1..n\}$ when obvious from the context. We write $\{f_j\}^{j \in \{1..n\}}$ or $\{f_j\}$ for the set $\{f_1, f_2, \cdots, f_n\}$. We write $x_1 :: \langle x_2, \ldots, x_n \rangle$ for the sequence

$\boxed{d, r, H, \varphi, b, P}$

Non-reactive behaviors	$d ::= x \mid c \mid f\ \langle d_i \rangle$	Phases	$\varphi \in \Phi ::= \varepsilon \mid \mathsf{later}$
Reactive behaviors	$r ::= \mathsf{init}\ x = c\ \mathsf{in}\ H$	Behaviors b	$::= d \mid r$
Event handlers	$H ::= \{I_i \Rightarrow d_i\ \varphi_i\}$	Programs P	$::= \{x_i = b_i\}$

$\boxed{FV(b)}$

$$FV(x) \equiv \{x\}, \quad FV(c) \equiv \emptyset, \quad FV(f\ \langle d_i \rangle) \equiv \bigcup_i FV(d_i)$$
$$FV(\mathsf{init}\ x = c\ \mathsf{in}\ \{I_i \Rightarrow d_i\ \varphi_i\}) \equiv \bigcup_i FV(d_i) - \{x\}$$

Fig. 2. Raw Syntax of E-FRP and definition of free variables

$\langle x_1, x_2, \ldots, x_n \rangle$. We write $A +\!\!+ A'$ for the concatenation of the sequences A and A'. We write $A \uplus B$ for $A \cup B$ assuming that $A \cap B = \emptyset$. Finally, we write $\mathrm{prim}(f, \langle c_i \rangle) \equiv c$ for that applying primitive function f on arguments $\langle c_i \rangle$ results in c.

2.1 Syntax of E-FRP

Figure 2 defines the syntax of E-FRP and the notion of free variables. The categories x, c and f are for variables, constants and primitive functions respectively. When a function symbol f is an infix operator, we write $d_1\ f\ d_2$ instead of $f\ d_1\ d_2$. The category I is for event names drawn from a finite set $\mathcal{I}$. On different target platforms, events may have different incarnations, like OS messages, or interrupts (as in the case of micro-controllers). For simplicity, an E-FRP event does not carry a value with its occurrence. The category ε stands for the empty string. An event I cannot occur more than once in an event handler H, and a variable x cannot be defined more than once in a program P.

A non-reactive behavior d is not directly updated by any event. Such a behavior could be a variable x, a constant c, or a function application on other non-reactive behaviors.

A reactive behavior $r \equiv \mathsf{init}\ x = c\ \mathsf{in}\ \{I_i \Rightarrow d_i\ \varphi_i\}$ initially has value c, and changes to the *current* value of d_i when event I_i occurs. Note that x is bound to the old value of r and can occur free in d_i. As mentioned earlier, the execution of an E-FRP program happens in two phases. Depending on whether φ_i is ε or later, the change of value takes place in either the first or the second phase.

A program P is just a set of mutually recursive behavior definitions.

2.2 Operational Semantics

A store S maps variables to values:

$$S ::= \{x_i \mapsto c_i\}$$

Figure 3 defines the operational semantics of an E-FRP program by means of four judgments. The judgments can be read as follows:

$\boxed{P \vdash b \xrightarrow{I} c}$

$$\frac{P \vdash b \xrightarrow{I} c}{P \uplus \{x = b\} \vdash x \xrightarrow{I} c} \qquad \frac{}{P \vdash c \xrightarrow{I} c}$$

$$\frac{\left\{P \vdash d_i \xrightarrow{I} c_i\right\} \quad \mathrm{prim}(f, \langle c_i \rangle) \equiv c}{P \vdash f\ \langle d_i \rangle \xrightarrow{I} c}$$

$$\frac{P \vdash d[x := c] \xrightarrow{I} c'}{P \vdash \mathsf{init}\ x = c\ \mathsf{in}\ \{I \Rightarrow d\} \uplus H \xrightarrow{I} c'}$$

$$\frac{H \not\equiv \{I \Rightarrow d\} \uplus H'}{P \vdash \mathsf{init}\ x = c\ \mathsf{in}\ H \xrightarrow{I} c}$$

$\boxed{P \vdash b \xrightarrow{I} b'}$

$$\frac{}{P \vdash d \xrightarrow{I} d}$$

$$\frac{H \equiv \{I \Rightarrow d\ \varphi\} \uplus H' \quad P \vdash d[x := c] \xrightarrow{I} c'}{P \vdash \mathsf{init}\ x = c\ \mathsf{in}\ H \xrightarrow{I} \mathsf{init}\ x = c'\ \mathsf{in}\ H}$$

$$\frac{H \not\equiv \{I \Rightarrow d\ \varphi\} \uplus H'}{P \vdash \mathsf{init}\ x = c\ \mathsf{in}\ H \xrightarrow{I} \mathsf{init}\ x = c\ \mathsf{in}\ H}$$

$\boxed{P \vdash b \xrightarrow{I} c; b'}$

$$\frac{P \vdash b \xrightarrow{I} c \quad P \vdash b \xrightarrow{I} b'}{P \vdash b \xrightarrow{I} c; b'}$$

$\boxed{P \xrightarrow{I} S; P'}$

$$\frac{\left\{\{x_i = b_i\}^{i \in K} \vdash b_j \xrightarrow{I} c_j; b'_j\right\}^{j \in K}}{\{x_i = b_i\}^{i \in K} \xrightarrow{I} \{x_i \mapsto c_i\}^{i \in K}; \{x_i = b'_i\}^{i \in K}}$$

Fig. 3. Operational Semantics of E-FRP

- $P \vdash b \xrightarrow{I} c$: "on event I, behavior b yields c."
- $P \vdash b \xrightarrow{I} b'$: "on event I, behavior b is updated to b'."
- $P \vdash b \xrightarrow{I} c; b'$: "on event I, behavior b yields c, and is updated to b'." This is shorthand for $P \vdash b \xrightarrow{I} c$ and $P \vdash b \xrightarrow{I} b'$.
- $P \xrightarrow{I} S; P'$: "on event I, program P yields store S and is updated to P'."

Note the difference between phase 1 stepping ($\varphi \equiv \varepsilon$) and phase 2 stepping ($\varphi \equiv \mathsf{later}$): The first returns the updated state, while the second returns the old state.

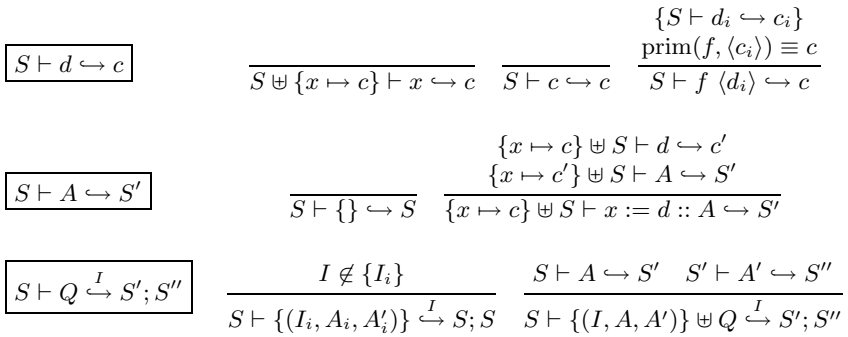

Fig. 4. Operational Semantics of SimpleC

3 SimpleC: An Imperative Language

The PIC16C66 does not need to be programmed in assembly language, as there is a commercial compiler that takes as input a restricted C. We compile E-FRP to a simple imperative language we call SimpleC, from which C or almost any other imperative language code can be trivially generated.

3.1 The Syntax of SimpleC

The syntax of SimpleC is defined as follows:

$$\text{Assignment Sequences} \quad A ::= \langle x_i := d_i \rangle \qquad \text{Programs} \quad Q ::= \{(I_i, A_i, A_i')\}$$

A program Q is just a collection of event handler function definitions, where I_i is the name of the function. The function body is split into two consecutive parts A_i and A_i', which are called *the first phase* and *the second phase* respectively. The idea is that when an interrupt I_i occurs, first the associated A_i is executed to generate a store matching the store yielded by the source program, then the associated A_i' is executed to prepare the store for the next event.

3.2 Operational Semantics of SimpleC

Figure 4 gives an operational semantics to SimpleC, where the judgments are read as follows:

- $S \vdash d \hookrightarrow c$: "under store S, d evaluates to c."
- $S \vdash A \hookrightarrow S'$: "executing assignment sets A updates store S to S'."
- $S \vdash Q \overset{I}{\hookrightarrow} S'; S''$: "when event I occurs, program Q updates store S to S' in the first phase, then to S'' in the second phase."

It is obvious that the operational semantics of SimpleC is deterministic.

4 Compilation

A compilation strategy is described by a set of judgments presented in Figure 5. The judgments are read as follows:

- $(x := d) < A$: "d does not depend on x or any variable updated in A."
- $\vdash_I^1 A : P$: "A is the first phase of P's event handler for I."
- $\vdash_I^2 A : P$: "A is the second phase of P's event handler for I."
- $P \rightsquigarrow Q$: "P compiles to Q."

In the object program, besides allocating one variable for each behavior defined in the source program, we allocate a *temporary variable* x^+ for each reactive behavior named x. Temporary variables are needed for compiling certain recursive definitions.

The compilation relation is clearly decidable. But note that given a program P, $P \rightsquigarrow Q$ does not uniquely determine Q. Hence, this is not a just specification of our compiler (which is deterministic), but rather, it allows many more compilers than the one we have implemented. However, we can prove that *any* Q satisfying $P \rightsquigarrow Q$ will behave in the same way.

Note also that if there is cyclic data dependency in an E-FRP program P, we will not be able to find a Q such that $P \rightsquigarrow Q$. The restriction that the set of possible events is known at compile time and the events are mutually exclusive (i.e. no two events can be active at the same time) allows us to perform this check.

Since the target code only uses fixed number of variables, has finite number of assignments, has no loops, and does not dynamically allocate space, it is obvious that the target code can be executed in bounded space and time.

4.1 Compilation Examples

The workings of the compilation relation are best illustrated by some examples. Consider the following source program:

$$x_1 = \mathsf{init}\ x = 0\ \mathsf{in}\ \{I_1 \Rightarrow x + x_2\},$$
$$x_2 = \mathsf{init}\ y = 1\ \mathsf{in}\ \{I_2 \Rightarrow y + x_1\}$$

Here x_1 depends on x_2, and x_2 depends on x_1, but they only depend on each other in different events. Within each event, there is no cyclic dependency. Hence this program can be compiled to the following SimpleC program:

$$(I_1, \langle x_1 := x_1^+ + x_2\rangle, \langle x_1^+ := x_1\rangle),$$
$$(I_2, \langle x_2 := x_2^+ + x_1\rangle, \langle x_2^+ := x_2\rangle)$$

Consider the following source program:

$$x_1 = \mathsf{init}\ x = 0\ \mathsf{in}\ \{I \Rightarrow x + x_2\},$$
$$x_2 = \mathsf{init}\ y = 1\ \mathsf{in}\ \{I \Rightarrow x_1\ \mathsf{later}\}$$

$\boxed{(x := d) < A}$

$$\frac{FV(d) \cap (\{x\} \uplus \{x_i\}) \equiv \emptyset}{(x := d) < \langle x_i := d_i \rangle}$$

$\boxed{\vdash_I^1 A : P}$

$$\frac{}{\vdash_I^1 \langle\rangle : \{\}} \qquad \frac{\vdash_I^1 A : P \quad (x := d) < A}{\vdash_I^1 x := d :: A : \{x = d\} \uplus P}$$

$$\frac{\vdash_I^1 A : P \quad (x := d[y := x^+]) < A}{\vdash_I^1 x := d[y := x^+] :: A : \{x = \mathsf{init}\ y = c\ \mathsf{in}\ \{I \Rightarrow d\} \uplus H\} \uplus P}$$

$$\frac{\vdash_I^1 A : P \quad (x^+ := d[y := x]) < A}{\vdash_I^1 x^+ := d[y := x] :: A : \{x = \mathsf{init}\ y = c\ \mathsf{in}\ \{I \Rightarrow d\ \mathsf{later}\} \uplus H\} \uplus P}$$

$$\frac{\vdash_I^1 A : P \quad H \not\equiv \{I \Rightarrow d\ \varphi\} \uplus H'}{\vdash_I^1 A : \{x = \mathsf{init}\ y = c\ \mathsf{in}\ H\} \uplus P}$$

$\boxed{\vdash_I^2 A : P}$

$$\frac{}{\vdash_I^2 \langle\rangle : \{\}} \qquad \frac{\vdash_I^2 A : P \quad (x := d) < A}{\vdash_I^2 x := d :: A : \{x = d\} \uplus P}$$

$$\frac{\vdash_I^2 A : P}{\vdash_I^2 x^+ := x :: A : \{x = \mathsf{init}\ y = c\ \mathsf{in}\ \{I \Rightarrow d\} \uplus H\} \uplus P}$$

$$\frac{\vdash_I^2 A : P}{\vdash_I^2 x := x^+ :: A : \{x = \mathsf{init}\ y = c\ \mathsf{in}\ \{I \Rightarrow d\ \mathsf{later}\} \uplus H\} \uplus P}$$

$$\frac{\vdash_I^2 A : P \quad H \not\equiv \{I \Rightarrow d\ \varphi\} \uplus H'}{\vdash_I^2 A : \{x = \mathsf{init}\ y = c\ \mathsf{in}\ H\} \uplus P}$$

$\boxed{P \rightsquigarrow Q}$

$$\frac{P \equiv \{x_i = b_i\} \quad \{\vdash_I^1 A_I : P \quad \vdash_I^2 A'_I : P\}^{I \in \mathcal{I}} \quad \bigcup_i FV(b_i) \subseteq \{x_i\}}{P \rightsquigarrow \{(I, A_I, A'_I)\}^{I \in \mathcal{I}}}$$

Fig. 5. Compilation of E-FRP

Here x_1 and x_2 are mutually dependent in I, but their values are updated in different phases. In each phase, the dependency graph is acyclic. Hence this program compiles too, and one acceptable compilation for it is:

$$(I, \langle x_1 := x_1^+ + x_2;\ x_2^+ := x_1 \rangle, \langle x_1^+ := x_1;\ x_2 := x_2^+ \rangle)$$

However, if the definition for x_2 were

$$x_2 = \mathsf{init}\ y = 1\ \mathsf{in}\ \{I \Rightarrow x_1\}$$

then the code would have been rejected.

If we let both x_1 and x_2 be updated in the later phase, i.e. let the program P be

$$x_1 = \mathsf{init}\ x = 0\ \mathsf{in}\ \{I \Rightarrow x + x_2\ \mathsf{later}\},$$
$$x_2 = \mathsf{init}\ y = 1\ \mathsf{in}\ \{I \Rightarrow x_1\ \mathsf{later}\}$$

then one possible translation of P in SimpleC is

$$(I, \langle x_1^+ := x_1 + x_2;\ x_2^+ := x1\rangle, \langle x_1 := x_1^+;\ x_2 := x_2^+\rangle$$

In concrete C syntax, the code would look like

```
void on_I( void ) {
        x1_ = x1 + x2;  x2_ = x1;
later:  x1  = x1_;      x2  = x2_;
}
```

4.2 Correctness

When an event I occurs, a behavior x in an E-FRP program can be updated to have a new value c. To prove that our compiler is correct, we need to show that after executing the event handler for I, the corresponding variable(s) of x in the target program will have the same value c. The following concept is useful for formalizing this intuition:

Definition 1 The **state** of a program P, written as $\text{state}(P)$, is a store defined by:

$$\text{state}(P) \equiv \{x_i \mapsto \text{state}_P(d_i)\} \uplus \{x_j \mapsto \text{state}_P(r_j), x_j^+ \mapsto \text{state}_P(r_j)\}$$

where $P \equiv \{x_i = d_i\} \uplus \{x_j = r_j\}$ and

$$\begin{array}{ll} \text{state}_{P \uplus \{x=b\}}(x) & \equiv \text{state}_P(b) \\ \text{state}_P(c) & \equiv c \\ \text{state}_P(f\ \langle d_i \rangle) & \equiv \text{prim}(f, \langle \text{state}_P(d_i) \rangle) \\ \text{state}_P(\mathsf{init}\ x = c\ \mathsf{in}\ H) & \equiv c \end{array}$$

Finally, we show that the compilation is correct in the sense that updating an E-FRP program does not change its translation in SimpleC, and that the source program generates the same result as the target program:

Theorem 1 (*Compilation Correctness*)

1. $P \rightsquigarrow Q \wedge P \xrightarrow{I} S; P' \implies P' \rightsquigarrow Q$;
2. $P \rightsquigarrow Q \wedge P \xrightarrow{I} S; P' \wedge \text{state}(P) \vdash Q \overset{I}{\hookrightarrow} S_1; S_2 \implies \exists S'.\ S_1 \equiv S \uplus S' \wedge S_2 \equiv \text{state}(P')$.

The proof of Theorem 1 is omitted as this paper focuses on practical rather than theoretical results.

$$\boxed{P \vdash Q \Rightarrow_1 Q'} \quad \frac{(x = d) \in P \quad FV(d) \cap LV(A) = \emptyset}{P \vdash Q \uplus \{(I, A +\!\!+ x := d +\!\!+ A', A'')\} \Rightarrow_1 Q \uplus \{(I, A +\!\!+ A', A'')\}}$$

$$\frac{(x = d) \in P \quad FV(d) \cap LV(A') = \emptyset}{P \vdash Q \uplus \{(I, A, A' +\!\!+ x := d +\!\!+ A'')\} \Rightarrow_1 Q \uplus \{(I, A, A' +\!\!+ A'')\}}$$

$$\boxed{P \vdash Q \Rightarrow_2 Q'} \quad \frac{x^+ \in FV(d)}{\begin{array}{c} P \vdash Q \uplus \{(I, A +\!\!+ x := d +\!\!+ A', A'')\} \Rightarrow_2 \\ Q \uplus \{(I, A +\!\!+ x := d[x^+ := x] +\!\!+ A', A'')\} \end{array}}$$

$$\boxed{P \vdash Q \Rightarrow_3 Q'} \quad \frac{FV(d) \cap LV(A_2) = \emptyset}{\begin{array}{c} P \vdash Q \uplus \{(I, A_1 +\!\!+ x^+ := d +\!\!+ A_2, A_3 +\!\!+ x := x^+ +\!\!+ A_4)\} \Rightarrow_3 \\ Q \uplus \{(I, A_1 +\!\!+ A_2 +\!\!+ x := d, x^+ := x +\!\!+ A_3 +\!\!+ A_4)\} \end{array}}$$

$$\boxed{P \vdash Q \Rightarrow_4 Q'} \quad \frac{Q \equiv Q' \uplus \{(I, A +\!\!+ x^+ := d +\!\!+ A', A'')\} \quad x^+ \notin RV(Q)}{P \vdash Q \Rightarrow_4 Q' \uplus \{(I, A +\!\!+ A', A'')\}}$$

$$\frac{Q \equiv Q' \uplus \{(I, A, A' +\!\!+ x^+ := d +\!\!+ A'')\} \quad x^+ \notin RV(Q)}{P \vdash Q \Rightarrow_4 Q' \uplus \{(I, A, A' +\!\!+ A'')\}}$$

$$\boxed{P \vdash Q \Rightarrow_i^* Q'} \quad \frac{}{P \vdash Q \Rightarrow_i^* Q} \qquad \frac{P \vdash Q \Rightarrow_i^* Q' \quad P \vdash Q' \Rightarrow_i Q''}{P \vdash Q \Rightarrow_i^* Q''}$$

$$\boxed{P \vdash Q \Rightarrow|_i Q'} \quad \frac{P \vdash Q \Rightarrow_i^* Q' \quad \nexists Q''.P \vdash Q' \Rightarrow_i Q''}{P \vdash Q \Rightarrow|_i Q'}$$

$$\boxed{P \vdash Q \Rightarrow| Q'} \quad \frac{\begin{array}{c} P \rightsquigarrow Q \quad P \vdash Q \Rightarrow|_1 Q_1 \quad P \vdash Q_1 \Rightarrow|_2 Q_2 \\ P \vdash Q_2 \Rightarrow|_3 Q_3 \quad P \vdash Q_3 \Rightarrow|_4 Q' \end{array}}{P \vdash Q \Rightarrow| Q'}$$

Fig. 6. Optimization of the Target Code

5 Optimization

The compiler that we have described, although provably correct, generates only naïve code that leaves a lot of room for optimization. Besides applying known techniques for optimizing sequential imperative programs, we can improve the target code by taking advantage of our knowledge on the compiler. In particular, we implemented a compiler that does:

1. Ineffective code elimination: Given a definition $x = b$ in the source program, x is affected by a particular event I only when one of the following is true:
   (a) b is a reactive behavior that reacts to I; or
   (b) b is a non-reactive behavior, and one of the variables in $FV(b)$ is affected by I.

   It is obvious that whether x is affected by I can be determined statically. If x is not affected by I, the code for updating x in the event handler for I

can be eliminated. This optimization helps reduce the code size as well as the response time.

2. Temporary variable elimination: The value of a temporary variable x^+ cannot be observed by the user of the system. Hence there is no need to keep x^+ around if it can be eliminated by some program transformations. This technique reduces memory usage and the response time.

We define the *right-hand-side variables* (written RV) of a collection of assignments, and the *left-hand-side variables* (written LV) of a collection of assignments, as

$$\boxed{RV(A)} \qquad RV(\langle x_i := d_i \rangle) \equiv \bigcup_i FV(d_i)$$

$$\boxed{RV(Q)} \qquad RV(\{(I_i, A_i, A_i')\}) \equiv \bigcup_i (RV(A_i) \cup RV(A_i'))$$

$$\boxed{LV(A)} \qquad LV(\langle x_i := d_i \rangle) \equiv \{x_i\}$$

The optimizations are carried out in four stages, where each stage consists of a sequence of one particular kind of transformations. In each stage, we keep applying the same transformation to the target code till it cannot be applied any further.

The purpose of stage 1 is to eliminate unnecessary updates to variables. One such update is eliminated in each step.

The code produced by the unoptimizing compiler has the property that at the beginning of phase 1 of any event handler, x and x^+ always hold the same value. Stage 1 optimization preserves this property. Hence in stage 2, we can safely replace the occurrences of x^+ on the right hand side of an assignment in phase 1 with x. This reduces the usage of x^+ and thus helps stage 4.

Since stage 1 and stage 2 are simple, we can easily write a compiler that directly generates code for stage 3. However the proof of the correctness of this partially-optimizing compiler then becomes complicated. Therefore we choose to present stage 1 and stage 2 as separate optimization processes.

We call the transformation in stage 3 "castling" for its resemblance to the castling special move in chess: if an event handler updates x^+ in phase 1 and assigns x^+ to x in phase 2, under certain conditions we can instead update x at the end of phase 1 and assign x to x^+ at the beginning of phase 2. This transformation reduces right-hand-side occurrences of x^+, thus making it easier to be eliminated in stage 4. Note that the correctness of this transformation is non-trivial.

If the value of a temporary variable x^+ is never used, then there is no need to keep x^+ around. This is what we do in stage 4.

Figure 6 formally defines the optimization. Given a source program P, we write $P \vdash Q \Rightarrow_i Q'$ for that Q is transformed to Q' in *one step* in stage i, where $1 \leq i \leq 4$; we write $P \vdash Q \Rightarrow_i^* Q'$ for that Q is transformed to Q' in *zero or more steps* in stage i; and we write $P \vdash Q \Rightarrow|_i Q'$ for that Q' is the *furthest* you can get from Q by applying stage i transformations. Finally, we write $P \vdash Q \Rightarrow| Q'$ for that P compiles to Q, which is then optimized to Q'.

$$
\begin{array}{l}
Q_1 \equiv (\mathit{IncSpd},\ \langle ds := ds^+ + 1\rangle, \langle ds^+ := ds\rangle),\\
\quad (\mathit{DecSpd},\ \langle ds := ds^+ - 1\rangle, \langle ds^+ := ds\rangle),\\
\quad (\mathit{Timer}1,\ \langle s^+ := 0; dc := \mathsf{if}\ dc^+ < 100\ \mathsf{and}\ s < ds\ \mathsf{then}\ dc^+ + 1\\
\qquad\qquad\qquad\qquad \mathsf{else\ if}\ dc^+ > 0\ \mathsf{and}\ s > ds\ \mathsf{then}\ dc^+ - 1\ \mathsf{else}\ dc^+;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0\rangle,\\
\qquad\qquad \langle s := s^+;\ dc^+ := dc\rangle),\\
\quad (\mathit{Stripe},\ \ \langle s := s^+ + 1;\rangle, \langle s^+ := s;\rangle),\\
\quad (\mathit{Timer}0,\ \langle count := \mathsf{if}\ count^+ >= 100\ \mathsf{then}\ 0\ \mathsf{else}\ count^+ + 1;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0\rangle,\\
\qquad\qquad \langle count^+ := count\rangle)
\end{array}
$$

$$
\begin{array}{l}
Q_2 \equiv (\mathit{IncSpd},\ \langle ds := ds + 1\rangle, \langle ds^+ := ds\rangle),\\
\quad (\mathit{DecSpd},\ \langle ds := ds - 1\rangle, \langle ds^+ := ds\rangle),\\
\quad (\mathit{Timer}1,\ \langle s^+ := 0; dc := \mathsf{if}\ dc < 100\ \mathsf{and}\ s < ds\ \mathsf{then}\ dc + 1\\
\qquad\qquad\qquad\qquad \mathsf{else\ if}\ dc > 0\ \mathsf{and}\ s > ds\ \mathsf{then}\ dc - 1\ \mathsf{else}\ dc;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0\rangle,\\
\qquad\qquad \langle s := s^+;\ dc^+ := dc\rangle),\\
\quad (\mathit{Stripe},\ \ \langle s := s + 1;\rangle, \langle s^+ := s;\rangle),\\
\quad (\mathit{Timer}0,\ \langle count := \mathsf{if}\ count >= 100\ \mathsf{then}\ 0\ \mathsf{else}\ count + 1;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0\rangle,\\
\qquad\qquad \langle count^+ := count\rangle)
\end{array}
$$

$$
\begin{array}{l}
Q_3 \equiv (\mathit{IncSpd},\ \langle ds := ds + 1\rangle, \langle ds^+ := ds\rangle),\\
\quad (\mathit{DecSpd},\ \langle ds := ds - 1\rangle, \langle ds^+ := ds\rangle),\\
\quad (\mathit{Timer}1,\ \langle dc := \mathsf{if}\ dc < 100\ \mathsf{and}\ s < ds\ \mathsf{then}\ dc + 1\\
\qquad\qquad\qquad\quad \mathsf{else\ if}\ dc > 0\ \mathsf{and}\ s > ds\ \mathsf{then}\ dc - 1\ \mathsf{else}\ dc;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0; s := 0\rangle,\\
\qquad\qquad \langle s^+ := s;\ dc^+ := dc\rangle),\\
\quad (\mathit{Stripe},\ \ \langle s := s + 1;\rangle, \langle s^+ := s;\rangle),\\
\quad (\mathit{Timer}0,\ \langle count := \mathsf{if}\ count >= 100\ \mathsf{then}\ 0\ \mathsf{else}\ count + 1;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0\rangle,\\
\qquad\qquad \langle count^+ := count\rangle)
\end{array}
$$

$$
\begin{array}{l}
Q_4 \equiv (\mathit{IncSpd},\ \langle ds := ds + 1\rangle, \langle\rangle),\\
\quad (\mathit{DecSpd},\ \langle ds := ds - 1\rangle, \langle\rangle),\\
\quad (\mathit{Timer}1,\ \langle dc := \mathsf{if}\ dc < 100\ \mathsf{and}\ s < ds\ \mathsf{then}\ dc + 1\\
\qquad\qquad\qquad\quad \mathsf{else\ if}\ dc > 0\ \mathsf{and}\ s > ds\ \mathsf{then}\ dc - 1\ \mathsf{else}\ dc;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0; s := 0\rangle, \langle\rangle),\\
\quad (\mathit{Stripe},\ \ \langle s := s + 1;\rangle, \langle\rangle),\\
\quad (\mathit{Timer}0,\ \langle count := \mathsf{if}\ count >= 100\ \mathsf{then}\ 0\ \mathsf{else}\ count + 1;\\
\qquad\qquad output := \mathsf{if}\ count < dc\ \mathsf{then}\ 1\ \mathsf{else}\ 0\rangle, \langle\rangle)
\end{array}
$$

Fig. 7. Optimization of the Simple RoboCup Controller

As an example, Figure 7 presents the intermediate and final results of optimizing the SRC program. Stage 1 optimization generates Q_1, where all unnecessary updates to *output* have been eliminated. Then stage 2 gets rid of all right-hand-side temporary variables in phase 1, as in the change from $ds := ds^+ + 1$

to $ds := ds + 1$, and the result is Q_2. In stage 3 we rearrange the updating to s and s^+ and get Q_3. Finally, we are able to remove all temporary variables in stage 4, resulting in Q_4, the optimized final code.

6 Discussion and Related Work

We refer the reader to [17] for a general introduction on the use of functional programming for reactive (especially real-time) applications.

E-FRP is a slightly extended subset of FRP. To make the relation precise, we present a partial function $[\![\_]\!]$ that translates E-FRP to FRP:

$$
\begin{array}{lll}
[\![x]\!] & \equiv x \\
[\![c]\!] & \equiv \mathsf{lift0}\ c \\
[\![f\ \langle d_i \rangle^{i \in \{1..n\}}]\!] & \equiv \mathsf{lift}n\ f\ \langle [\![d_i]\!] \rangle^{i \in \{1..n\}} \\
[\![\mathsf{init}\ x = c\ \mathsf{in}\ \{I_i \Rightarrow d_i\}]\!] & \equiv \mathsf{iStepAccum}\ c\ [\![\{I_i \Rightarrow d_i\}]\!]_x \\
[\![\mathsf{init}\ x = c\ \mathsf{in}\ \{I_i \Rightarrow d_i\ \mathsf{later}\}]\!] & \equiv \mathsf{stepAccum}\ c\ [\![\{I_i \Rightarrow d_i\}]\!]_x \\
[\![\{x_i = b_i\}]\!] & \equiv \{x_i = [\![b_i]\!]\} \\
\quad \text{where} \quad [\![\{I_i \Rightarrow d_i\}^{i \in K}]\!]_x & \equiv \mathsf{foldr}\ (.|.)\ \mathsf{neverE}\ [[\![d_i]\!]_{x,I_i}]^{i \in K} \\
[\![d]\!]_{x,I} & \equiv (\mathsf{snapshot}n\ I\ \langle x_i \rangle^{i \in \{1..n\}}) \Rightarrow \lambda \langle x_i \rangle^{i \in \{1..n\}}.\lambda x.d \\
& \qquad \text{where } \{x_i\}^{i \in \{1..n\}} \equiv FV(d) - \{x\}
\end{array}
$$

The key facility missing from FRP is the ability to mix now and later operations, which is exactly the reason why $[\![\_]\!]$ is not total. Our experience with E-FRP shows that this is a very useful feature, and extending FRP with such functionality would be interesting future work.

Several languages have been proposed around the *synchronous data-flow* notion of computation, including SIGNAL, LUSTRE, and ESTEREL, which were specifically designed for control of real-time systems. SIGNAL [8] is a block-diagram oriented language, where the central concept is a *signal*, a time-ordered sequence of values. This is analogous to the sequence of values generated in the execution of an E-FRP program. LUSTRE [3] is a functional synchronous data flow language, rooted again in the notion of a sequence. ESTEREL is a synchronous imperative language devoted to programming control-dominated software or hardware reactive systems [1]. Compilers exist to translate ESTEREL programs into automata or electronic circuits [2]. Although these languages are similar to E-FRP, there are subtle differences whose significance deserves further investigation.

To guarantee bounded space and time, the languages above do not consider recursion. The language of synchronous Kahn networks [4] was developed as an extension to LUSTRE that adds recursion and higher-order programming. Such an extension yields a large increase in expressive power, but sacrifices resource-boundedness. In RT-FRP we have shown that, using some syntactic restrictions and a type system, it is possible to have recursion and guarantee resource bound. We expect to be able to extend E-FRP with the RT-FRP style recursion.

The SCR (Software Cost Reduction) requirements method [10] is a formal method based on tables for specifying the requirements of safety-critical software

systems. SCR is supported by an integrated suite of tools called SCR* [11], which includes among others a *consistency checker* for detecting well-formedness errors, a *simulator* for validating the specification, and a *model checker* for checking application properties. SCR has been successfully applied in some large projects to expose errors in the requirements specification. We have noticed that the tabular notation used in SCR for specifying system behaviors has a similar flavor as that of E-FRP, and it would be interesting future work to translate E-FRP into SCR and hence leverage the SCR* toolset to find errors in the program.

Throughout our work on E-FRP, more systematic approaches for semantics-based program generation, for example MetaML, have been continually considered for the implementation of the final program generation phase of the compiler [18]. So far, we have used an elementary technique for the generation of programs instead. There are a number of practical reasons for this:

- The code we generate does not involve the generation of new variable names, and thus there is no apparent need for MetaML hygiene (fresh name generation mechanism).
- MetaML does not currently support generating code in languages other than MetaML.
- The target language, has only a very simple type system, and essentially all the instructions we generated have the same type. In other words, assuring type safety of the generated code, a key feature of MetaML, is not a significant issue.
- The standard way for using MetaML is to stage a denotational semantics. While such a denotational semantics exists for FRP, it is not yet clear how we can use staging to introduce the imperative implementation of behaviors in a natural way. We plan to pursue this direction in future work.

An alternative approach to semantics-based program generation is the use of an appropriate monad, as in the work of Harrison and Kamin [9]. However, we have not identified such a monad at this point, and this is also interesting future work.

7 Conclusions

We have presented a core, first-order subset of FRP and showed how it can be compiled in a natural and sound manner into a simple imperative language. We have implemented this compilation strategy in Haskell. The prototype produces C code that is readily accepted by the PIC16C66 C compiler. Compared to previous work on RT-FRP, our focus here has shifted from the integration with the lambda language to the scrutiny of the feasibility of implementing this language and using it to program real event-driven systems. In the immediate future, we intend to extend the language with more constructs (especially behavior switching), to investigate the correctness of some basic optimizations on the generated programs, and to continue the development of the compiler.

References

1. Gerard Berry and Laurent Cosserat. The Esterel synchronous programming language and its mathematical semantics. In A.W. Roscoe S.D. Brookes and editors G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lect. Notes in Computer Science*, pages 389–448. Springer Verlag, 1985.
2. Gerard Berry and the Esterel Team. *The Esterel v5.21 System Manual*. Centre de Mathématiques Appliquées, Ecole des Mines de Paris and INRIA, March 1999. Available at `http://www.inria.fr/meije/esterel`.
3. Paul Caspi, Halbwachs Halbwachs, Nicolas Pilaud, and John A. Plaice. Lustre: A declarative language for programming synchronous systems. In *the Symposium on Principles of Programming Languages (POPL '87)*, January 1987.
4. Paul Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*. ACM SIGPLAN, May 1996.
5. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.
6. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
7. John Peterson et al. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, Mar 1997. World Wide Web version at `http://haskell.cs.yale.edu/haskell-report`.
8. Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lect Notes in Computer Science, edited by G. Goos and J. Hartmanis*, pages 257–277. Springer-Verlag, 1987.
9. William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE International Conference on Computer Languages*, 1998.
10. Constance Heitmeyer. Applying *Practical* formal methods to the specification and analysis of security properties. In *Proc. Information Assurance in Computer Networks, LNCS 2052*, St. Petersburg, Russia, May 2001. Springer-Verlag.
11. Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification*, Vancouver, Canada, 1998.
12. Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.
13. Microchip Technology Inc. *PIC16C66 Datasheet*. Available on-line at `http://www.microchip.com/`.
14. John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.
15. Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *Proceedings of International Conference on Software Engineering*, May 1999.
16. RoboCup official site. `http://www.robocup.org/`.
17. Walid Taha, Paul Hudak, and Zhanyong Wan. Directions in functional programming for real(-time) applications. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Proc. First International Workshop, EMSOFT, LNCS 2211*, pages 185–203, Tahoe City, CA, USA, October 2001. Springer-Verlag.

18. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of Symposium on Partial Evaluation and Semantics Based Program manipulation*, pages 203–217. ACM SIGPLAN, 1997.
19. Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN '00 COnference on Programming Language Design and Implementation (PLDI)*, pages 242–252, Vancouver, BC, Canada, June 2000. ACM, ACM Press.
20. Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of Sixth ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy, September 2001. ACM.