Abstract

Functional Reactive Programming for Real-Time Reactive Systems

Zhanyong Wan

2002

A real-time reactive system continuously reacts to stimuli from the environment by sending out responses, where each reaction must be made within certain time bound. As computers are used more often to control all kinds of devices, such systems are gaining popularity rapidly. However, the programming tools for them lag behind.

In this thesis we present RT-FRP, a language for real-time reactive systems. RT-FRP can be executed with guaranteed resource bounds, and improves on previous languages for the same domain by allowing a restricted form of recursive switching, which is a source of both expressiveness and computational cost. The balance between cost guarantee and expressiveness is achieved with a carefully designed syntax and type system.

To better suit hybrid systems and event-driven systems, we have designed two variants of RT-FRP called H-FRP and E-FRP respectively. We give H-FRP a continuous-time semantics natural for modeling hybrid systems and an operational semantics suitable for a discrete implementation. We show that under certain conditions the operational semantics converges to the continuous-time semantics as the sampling interval approaches zero. We also present a provably correct compiler for E-FRP, and use E-FRP to program both real and simulated robots.

Functional Reactive Programming

for Real-Time Reactive Systems

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Zhanyong Wan

Dissertation Director: Professor Paul Hudak

December 2002

Contents

List of Figures

Acknowledgments

First, I would like to thank my adviser, Paul Hudak, for his guidance and encouragement. Without his persistence and patience, I could not have made it through my Ph.D. program.

Special thanks to Walid Taha for his kind direction. This work owes much to his help even though he is not officially on my reading committee.

Many thanks to John Peterson for interesting discussions and insightful comments, and for constantly bugging me to go outdoors (thus keeping me alive through out my years at Yale).

Thanks to Zhong Shao for inspiration. Thanks to Conal Elliott for his early work on Fran, which has been the basis of my work.

This work has benefited from discussions with Henrik Nilsson, Antony Courtney, and Valery Trifonov.

Finally, I want to thank my family for their understanding and sacrifice.

Notations

Common:

$x = y$	x and y are semantically equal
$x \equiv y$	x and y are syntactically identical
$\mathbb{N}$	set of natural numbers, $\{1, 2, \cdots\}$
$\mathbb{N}_n$	set of the first n natural numbers, $\{1, 2, \cdots, n\}$ (for all $n \leq 0$, we define $\mathbb{N}_n$ to be the empty set)
$\mathbb{R}$	set of real numbers
$\mathbb{T}$	set of times (non-negative real numbers, $\{t \in \mathbb{R} \mid t \geq 0\}$)
$A \to B$	function from A to B
$\lambda p.e$	function mapping bound pattern p to formula e
$\emptyset$	empty set, $\{\}$
$[a, b]$	closed interval, $\{x \in \mathbb{R} \mid a \leq x \leq b\}$
(a, b)	open interval, $\{x \in \mathbb{R} \mid a < x < b\}$
$\lvert r \rvert$	absolute value of real number r
$FV(E)$	set of free variables in term E
$E[x := F]$	substitution of all free occurrences of x for F in E
$\{m..n\}$	set of integers between m and n, $\{k \mid m \leq k \leq n\}$
$\langle m..n \rangle$	sequence of integers from m to n, $\langle m, m+1, \cdots, n \rangle$
$\{f_j\}^{j \in S}$	set of formulas where j is drawn from set S
$\langle f_j \rangle^{j \in S}$	sequence of formulas where j is drawn from sequence S

$\{f_1, f_2, \cdots, f_n\}$	set of formulas
$\langle f_1, f_2, \cdots, f_n \rangle$	sequence of formulas
$\{f_j\}$	set of formulas
$\langle f_j \rangle$	sequence of formulas
ε	empty term
$A \uplus B$	$A \cup B$ assuming that $A \cap B = \emptyset$
$A +\!\!+ B$	concatenation of sequences A and B
_	wildcard variable
$A \implies B$	A implies B
$\langle A \rangle$	the type "sequence of A"
$\bot$	bottom
$A_\bot$	lifted set A, $A \uplus \{\bot\}$
$\mid v - v' \mid$	difference between v and v'

For the BL language:

$()$	unit element or unit type
$(a_1, a_2, \cdots, a_n)$	n-tuple[1], where $n \geq 2$
Maybe α	option type whose values are either Nothing or Just v, where v has type α
True	synonym for Just $()$
False	synonym for Nothing
$f_1, \cdots, f_n : \alpha \to \alpha'$	$f_1, \cdots, f_n$ are functions from α to α'
Γ	variable context
$\Gamma(x) = \alpha$	context Γ assigns type α to x
$\Gamma \oplus \{x_j : \alpha_j\}$	context identical to Γ except that x_j is assigned type α_j
$\Gamma \vdash e : \alpha$	term e has type α in context Γ

[1]The notation of a 2-tuple is the same as an open interval, but the context always makes the intended meaning clear.

$v \in \alpha$	value v has type α in the empty context, $\emptyset \vdash v : \alpha$
$f\ v \hookrightarrow v'$	applying function f to value v yields v'
$\mathcal{X}$	variable environment
$\mathbb{X}$	set of all variable environments
$\mathcal{X}(x) = v$	environment $\mathcal{X}$ maps x to v
$\mathcal{X} \oplus \{x_j \mapsto v_j\}$	environment identical to $\mathcal{X}$ except that x_j maps to v_j
$\Gamma \vdash \mathcal{X}$	variable environment $\mathcal{X}$ is compatible with variable context Γ
$\mathcal{X} \vdash e \hookrightarrow v$	e evaluates to v in environment $\mathcal{X}$

For the RT-FRP language:

Θ	variable context
Δ	signal context
$\mathcal{Z}$	signal environment
$\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}} s : \alpha$	s is a signal of type α in contexts Γ, Θ, and Δ
$\Gamma;\ \Theta;\ \Delta \vdash \mathcal{Z}$	signal environment $\mathcal{Z}$ is compatible with contexts Γ, Θ, and Δ
$\mathcal{X} \vdash s \overset{t,i}{\hookrightarrow} v$	s evaluates to v on input i at time t, in environment $\mathcal{X}$
$\mathcal{X};\ \mathcal{Z} \vdash s \overset{t,i}{\longrightarrow} s'$	s is updated to become s' on input i at time t, in environment $\mathcal{X}$ and $\mathcal{Z}$
$\mathcal{X};\ \mathcal{Z} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v, s'$	shorthand for $\mathcal{X} \vdash s \overset{t,i}{\hookrightarrow} v$ and $\mathcal{X};\ \mathcal{Z} \vdash s \overset{t,i}{\longrightarrow} s'$
$s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v, s'$	shorthand for $\emptyset;\ \emptyset \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v, s'$
$\lvert s \rvert$	size of signal s
$\lVert \mathcal{Z} \rVert$	size bound of signals in $\mathcal{Z}$
$\lVert s \rVert_m$	size bound of signal s

For the H-FRP language:

Ω	behavior context
$\mathcal{B}$	behavior environment

$\mathbb{B}$	set of all behavior environments
$v_\perp$	optional value, v or $\perp$
$\mathcal{X}_\perp$	lifted variable environment, which maps variables to optional values
$\mathbb{X}_\perp$	set of all lifted variable environment
$\Omega \vdash \mathcal{B}$	behavior environment $\mathcal{B}$ is compatible with behavior contexts Ω
$\Gamma;\ \Omega \vdash\ b : \mathsf{B}\ \alpha$	b is a behavior of type α in context Γ and Δ
$\Gamma;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha$	ev is an event of type α in context Γ and Δ
$\mathbf{tr}[\![E]\!]$	translation of H-FRP term E to RT-FRP
$\mathbf{eval}[\![e]\!]_{\mathcal{X}}$	value of e in environment $\mathcal{X}$
P^t	time sequence whose last element is t
$\lvert P \rvert$	norm of time sequence P
$\lvert P \rvert_{t_0}$	norm of time sequence P with respect to start time t_0
$[v]_\epsilon$	ϵ-neighborhood of value v, $\{x \mid \lvert x - v \rvert < \epsilon\}$
$[S]_\epsilon$	ϵ-neighborhood of set S, $\Sigma_{v \in S}[v]_\epsilon$
$t \lessdot t'$	shorthand for either $t < t'$ or $t = t' = 0$
TS	set of time sequences, $\{\langle t_1, \cdots, t_n \rangle \mid n \in \mathbb{N} \text{ and } t_1 < \cdots < t_n\}$
Δt_j	shorthand for $t_{j+1} - t_j$
$\mathbf{at}[\![b]\!]_{\mathcal{B}}$	continuous semantics of behavior b in environment $\mathcal{B}$
$\widetilde{\mathrm{at}}[\![b]\!]_{\mathcal{B}}$	discrete semantics of behavior b in environment $\mathcal{B}$
$\widetilde{\mathrm{at}}^*[\![b]\!]_{\mathcal{B}}$	limit of discrete semantics of behavior b in environment $\mathcal{B}$
$\mathbf{occ}[\![ev]\!]_{\mathcal{B}}$	continuous semantics of event ev in environment $\mathcal{B}$
$\widetilde{\mathrm{occ}}[\![ev]\!]_{\mathcal{B}}$	discrete semantics of event ev in environment $\mathcal{B}$
$\widetilde{\mathrm{occ}}^*[\![ev]\!]_{\mathcal{B}}$	limit of discrete semantics of event ev in environment $\mathcal{B}$

For the E-FRP language:

$A +\!\!+ A'$	concatenation of sequences A and A'
$FV(d)$	set of free variables in d
$\Gamma \vdash d : \alpha$	d is a behavior of type α in context Γ
$\Gamma \vdash P$	Γ assigns types to behaviors in P
$P \vdash d \xrightarrow{I}\!\!\!\!\!\!\rightarrow v$	on event I, behavior d yields v
$P \vdash d \xrightarrow{I}\!\!\!\!\!\!\rightarrow d'$	on event I, behavior d is updated to d'
$P \vdash d \xrightarrow{I} v; d'$	shorthand for $P \vdash d \xrightarrow{I}\!\!\!\!\!\!\rightarrow v$ and $P \vdash d \xrightarrow{I}\!\!\!\!\!\!\rightarrow d'$
$P \xrightarrow{I} \mathcal{X}; P'$	on event I, program P yields environment $\mathcal{X}$ and is updated to P'
$\Gamma \vdash A$	assignment sequence A is compatible with Γ
$\Gamma \vdash \mathcal{X}; A$	environment $\mathcal{X}$ and assignment sequence A are compatible with Γ
$\Gamma \vdash \mathcal{X}; Q$	environment $\mathcal{X}$ and SimpleC program Q are compatible with Γ
$A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$	executing assignment sequence A updates environment $\mathcal{X}$ to $\mathcal{X}'$
$Q \vdash \mathcal{X} \overset{I}{\hookrightarrow} \mathcal{X}'; \mathcal{X}''$	when event I occurs, program Q updates environment $\mathcal{X}$ to $\mathcal{X}'$ in the first phase, then to $\mathcal{X}''$ in the second phase
$x := e < A$	e does not depend on x or any variable updated in A
$P \vdash^1_I A$	A is the first phase of P's event handler for I
$P \vdash^2_I A$	A is the second phase of P's event handler for I
$P \rightsquigarrow Q$	P compiles to Q
$RV(A)$	set of right-hand-side variables in assignments A
$RV(Q)$	set of right-hand-side variables in program Q
$LV(A)$	set of left-hand-side variables in assignments A
$P \vdash Q \Rightarrow\!\mid Q'$	P compiles to Q, and then is optimized to Q'
$\mathcal{X}\mid X$	environment $\mathcal{X}$ restricted to variables in set X, $\{x : \mathcal{X}(x) \mid x \in \mathrm{dom}(\mathcal{X}) \cap X\}$

Chapter 1

Introduction

1.1 Problem

According to Berry [5], computerized systems can be divided into three broad categories:

- *transformational systems* that compute output from inputs and then stop, examples including numerical computation and compilers,
- *interactive systems* that interact with their environments at a pace dictated by the computer systems, and
- *reactive systems* that continuously react to stimuli coming from their environment by sending back responses.

The main difference between an interactive system and a reactive system is that the latter is purely input-driven and must react at a pace dictated by the environment. Examples of reactive systems include signal processors, digital controllers, and interactive computer animation. A reactive system is said to be *real-time* if the responses must be made within a certain time bound or the system will fail. Today, many real-time reactive systems are being designed, implemented, and maintained, including aircraft engines, military robots, smart weapons, and many electronic gadgets. As this trend continues,

the reliability and safety of programming languages for such systems become more of a concern.

In our context, we call the sequences of time-stamped input, output, and intermediate, values in a reactive system *signals*. A signal can be viewed as the trace of a value that potentially changes with each reaction. Operations in a reactive system are often easier described as being performed on signals rather than on static values. For example, instead of updating the value of a variable, we often want to change the behavior of a whole signal; instead of adding two static values, we may need to add two signals pointwise. There are also some temporal operations that only make sense for signals, like integrating a numeric signal over time.

Programming real-time reactive systems presents some challenges atypical in the development of other systems:

1. Reactive systems emphasize signals rather than static values. Hence the programming languages should directly support the representation of signals and operations on them. As a signal is a potentially infinite sequence of time-stamped values, it cannot be directly represented in finite space, and thus requires special techniques.

2. Since the system has to respond to the environment fast enough to satisfy the real-time constraint, it is important to know at compile time the maximum amount of space and time a program might use to make a response.

3. Some reactive systems are *hybrid*, in the sense that they employ both components that work in discrete steps (e.g. event generators and digital computers) and components that work continuously (e.g. speedometers and numerical integrators). Continuous components must be turned into some form that can be handled by digital computers, for example using symbolic computation or by sampling techniques.

4. Furthermore, many such systems run on computers with limited space and computational power (e.g. embedded systems and digital signal processors), and hence

efficiency is a major concern.

1.2 Background

While programming languages and tools for transformational systems and interactive systems have been relatively well-studied, those for reactive systems have not been properly developed for historical reasons [5]. This section summarizes some previous attempts to program reactive systems.

1.2.1 General-purpose languages

Traditionally, reactive systems have been programmed using general-purpose imperative languages, C being a typical example. Unfortunately, this approach has some severe limitations:

- As the programming language lacks the ability to operate on signals as a whole, the programmer has to deal with snapshots of signals instead. Hence, a variable is allocated to hold the snapshot of a signal, and the program is responsible for updating this variable, possibly with each reaction. Great care must be taken by the programmer to ensure that a variable is referenced and updated at the right time, because the same variable is reused to hold different elements in a value sequence. Such low-level details make programming both tedious and error-prone.

- To complicate things even more, a dynamic reactive system may create or destroy signals at run time. When this happens, not only the memory associated with the signals needs to be allocated or reclaimed, the program also needs to know how to update the new signals with each reaction, and cease updating the dead signals. Again, the language provides no special support for this.

- As signals are not first-class in a general-purpose language, they cannot be stored in data structures or passed as function arguments or results, making programming

complex systems even more difficult.

- Finally, as the programming language is not designed with resource bounds in mind, it is usually difficult to analyze the space and time needed to make a reaction, not to mention guaranteed resource bounds. The reliability of real-time systems is thus compromised.

In summary, this approach provides the lowest level of abstraction and the least guarantee on resource consumption. Although the learning curve of this approach is the least steep for an ordinary programmer, development and maintenance cost is usually high while the software productivity and reliability are low. Hence general-purpose languages are not recommended except for trivial reactive systems.

1.2.2 Synchronous data-flow reactive languages

In a reactive system, signals may have different values depending on when they are observed. Hence we say "the value of expression e at time t" rather than simply "the value of expression e." Consider the case where we need to calculate the value of some expression $e_1 + e_2$ at time t. We can do this by calculating first e_1, then e_2, and then adding the results. Depending on how we treat time, there are two options:

1. We compute in real time: At time t, e_1 is evaluated. By the time this computation finishes, time has advanced to $t' > t$. We then evaluate e_2 at this later time t'. The final result will be the sum of the value of e_1 at t and the value of e_2 at t'. Clearly, as expressions get more complex, the meaning of them becomes increasingly dubious, and it is very hard to reason about a program or to perform optimizations.

2. Alternatively, we can assume that computations have zero duration. In other words, we "freeze" the logical time until the computation for one reaction step is done. In this framework, we will get the sum of the value of e_1 at t and the value of e_2 at t. This semantics is much easier to understand and work with, and is acceptable if computations can be shown to have bounded effective duration.

The second approach above is called a *synchronous* view on reactive systems.

In general, the value of a signal at time t may depend on the values of other signals at time t and/or the values of all signals before t (causality precludes it to depend on the value of some signal after t). However, a system is unimplementable if it requires remembering an unbounded length of history. For example, in the reactive system defined by

$$r_k = s_{\lfloor k/2 \rfloor}$$

where s_k is the k-th stimulus and r_k is the k-th response, we need to know the $\lfloor k/2 \rfloor$-th input as we are generating the k-th response. What's more, in order to be able to generate future responses, we need to remember the $(\lfloor k/2 \rfloor + 1)$-th to the k-th stimuli as well. As time advances, the buffer grows without bound. Hence this system cannot be implemented, at least with finite storage.

In our context, the word synchrony also means there is no need for an unbounded buffer to hold the history of signals. Hence the above system is not synchronous. As an example of synchronous systems, the Fibonacci sequence can be defined by

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 1 \\ a_{k+2} &= a_k + a_{k+1} \end{aligned}$$

which only needs a buffer to hold two previous values.

In the last two decades, several *synchronous data-flow* reactive programming languages, including SIGNAL[18], LUSTRE[7], and ESTEREL[5], have been proposed to develop reactive systems more easily and more reliably. Languages in this category assume that computations have zero duration and only bounded history is needed (hence the word "synchronous"), and manipulate signals as a whole rather than as individual snapshots (hence the name "data-flow").

These languages capture the synchronous characteristics of reactive systems, treat signals as first-class citizens, and thus are a vast improvement over general-purpose languages. However, in order to guarantee resource bounds, they allocate memory statically and are hence not flexible enough to naturally express systems that change functionalities at run time and thus need to allocate memory dynamically. Besides, they are built on the basis that time advances in discrete steps, and thus fail to provide adequate abstraction for continuous quantities.

1.2.3 FRP

Our work has been inspired by Functional Reactive Programming (FRP) [25, 53], a high-level declarative language for hybrid reactive systems. FRP was developed based on ideas from Elliott and Hudak's computer animation language Fran [15, 16]. It classifies signals into two categories: *behaviors* (values that continuously change over time) and *events* (values that are present only at discrete time points). The concept of a behavior is especially fitting for modeling a signal produced by a continuous component. Both behaviors and events are first-class, and a rich set of operations on them (including integration of numeric behaviors over time) are provided. An FRP program is a set of mutually recursive behaviors and events.

FRP has been used successfully in domains such as interactive computer animation [16], graphical user interface design [11], computer vision [44], robotics [42], and control systems [55], and is particularly suitable for hybrid systems. FRP is sufficiently high-level that, for many systems, FRP programs closely resemble equations naturally written by domain experts to specify the problems. For example, in the domain of control systems, a traditional mathematical specification of a controller can be transformed into running FRP code in a matter of minutes.

FRP is implemented as an embedded language [14, 24] in Haskell [17], and provides no guarantees on execution time or space: programs can diverge, consume large amounts

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

1.3 Goal

A good programming language for a certain problem domain should capture the essence of that domain and provide the right level of abstraction. Programs written in such a language should be close to the high-level specification of the problem. In particular, the programmer should not be concerned with implementation details that can be handled by the compiler, while at the same time he should be able to be specific about things he cares about. In the case of real-time reactive systems, this means the language should support the manipulation of signals naturally, provide formal assurance on resource bounds, and yield efficient programs.

This thesis presents our effort in developing such a language for real-time reactive systems. We call it RT-FRP for Real-Time Functional Reactive Programming [54]. We try to make the language simple for presentation purpose, yet interesting enough to convey all of our design ideas.

1.4 Approach

We call our language Real-Time FRP since it is roughly a subset of FRP. 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. By their very nature, RT-FRP programs do not terminate: they continuously emit values and interact with the environment. Thus it is appropriate to model RT-FRP program execution as an infinite sequence of steps. In each step, the current time and current stimuli are read, and the pro-

gram calculates an output value and updates its state. Our goal is to guarantee that every step executes in bounded time, and that overall program execution occurs in bounded space. We achieve this goal as follows:

1. Unlike FRP, which is embedded in Haskell, RT-FRP is designed to be a *closed language* [31]. This makes it possible to give a *direct* operational semantics to the language.

2. RT-FRP's operational semantics provides a well-defined and tractable notion of cost. That is, the size of the derivation for the judgment(s) defining a step of execution provides a measure of the amount of time and space needed to execute that step on a digital computer.

3. Recursion is a great source of expressiveness, but also of computational cost. A key contribution of our work is a restricted form of recursion in RT-FRP that limits both the time and space needed to execute such computations. We achieve this by first distinguishing two different kinds of recursion in FRP: one for feedback, and one for switching. Without constraint, the first form of recursion can lead to programs getting stuck, and the second form can cause terms to grow arbitrarily in size. We address these problems using carefully chosen syntax and a carefully designed type system. Our restrictions on recursive switching are inspired by tail-recursion [43].

4. We define a function $\|-\|$ on well-typed programs that determines an upper bound on the space and time resources consumed when executing a program. We prove that $\|s\| = O(2^{(\text{size of } s)})$ for a well-typed program s.

5. A key aspect of our approach is to split RT-FRP into two naturally distinguishable parts: a *reactive language* and a *base language*. Roughly speaking, the reactive language is comparable to a synchronous system [8], and the base language can be any resource-bounded pure language that we wish to extend to a reactive setting. We show that the reactive language is bounded in terms of both time and space, inde-

pendent of the base language. Thus we can reuse our approach with a new base language without having to re-establish these results. Real-time properties of the base language can be established independently, and such techniques already exist even for a functional base language [23, 28, 29, 36].

6. The operational semantics of RT-FRP can be used to guide its implementation, and we have developed an interpreter for RT-FRP in the Haskell language [26].

As with any engineering effort, the design of RT-FRP is about striking a balance between conflicting goals. In this case, we strive to make the language expressive yet resource bounded. On one end of the design spectrum we have FRP, which has full access to λ-calculus and therefore is by no means resource bounded; on the other end we have statically allocated languages like LUSTRE and ESTEREL, in which all necessary space has to be allocated at compile time. RT-FRP sits somewhere in the middle: it allows a restricted form of recursive switching, which incurs dynamic memory allocation, yet guarantees that programs only use bounded space and time.

However, like other synchronous data-flow languages, RT-FRP is based on the notion that time is discrete. While this is adequate for modeling systems where all components change states at discrete points in time, it is unnatural for *hybrid* reactive systems, which employ components that change continuously over time. For such systems, we developed a variant of RT-FRP called *Hybrid FRP* (H-FRP), in which we have both continuous-time-varying behaviors and discrete-time-occurring events.

In H-FRP, the best intuition for a behavior is a function from real-valued times. Such a function is defined at all times, and is free to change at any time. Hence, an H-FRP program is best modeled by a continuous-time-based denotational semantics. This semantics, although being handy for reasoning about the program, cannot be directly implemented on a digital computer, which by its very nature can only work discretely. Therefore we also define an operational semantics to guide a discrete implementation.

An interesting problem is that as the operational semantics works in discrete steps and thus is only an approximation to the continuous model, its result does not always match that given by the denotational semantics, although we expect them to be close. One of our major contribution is that, assuming that we can represent real numbers and carry out operations on them exactly, we prove that under suitable conditions the operational semantics converges to the continuous-time semantics as the sampling intervals approach zero, and we identify a set of such conditions. We should point out that in order to have this convergence property, H-FRP has to be more restricted than RT-FRP.

Some reactive systems are purely driven by events, where the system does not change unless an event occurs. In using RT-FRP for such event-driven systems, we found that the compiler can generate much more efficient code if we impose certain restrictions on the source program. We formalize these restrictions and call the restricted language *Event-driven FRP* (E-FRP) [55]. A compiler for E-FRP is defined and proved to be correct. While our use of RT-FRP and H-FRP is still experimental, we have successfully used E-FRP to program low-level controllers for soccer-playing robots and controllers for simulated robotic vehicles in the MindRover computer game [45].

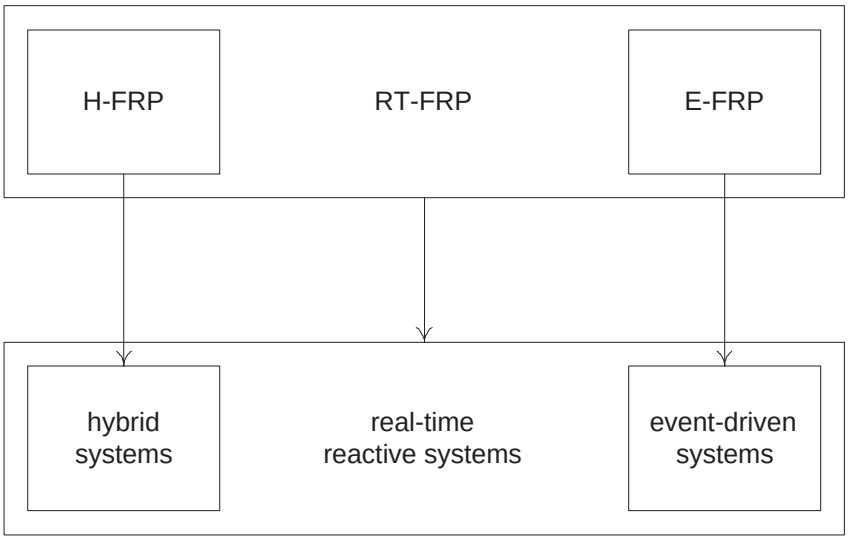


Figure 1.1: The RT-FRP family of languages and their domains

H-FRP and E-FRP are roughly subsets of RT-FRP (in fact, we give translations that turn H-FRP and E-FRP programs into RT-FRP). The hierarchy of the three languages and their corresponding domains are illustrated in Figure 1.1. An interesting observation is that the hierarchical relationship between the languages corresponds to that of the problem domains. This confirms our belief that nested domains call for nested domain-specific languages.

1.5 Contributions

Our contributions are fourfold: first, we have designed a new language RT-FRP and its two variants (H-FRP and E-FRP) that improve on previous languages; second, we have proved the resource bound property of programs written in RT-FRP; third, we have established the convergence property of H-FRP programs; and fourth, we have presented a provably-correct compiler for E-FRP.

More specifically, our contributions toward language design are: a modular way of designing a reactive language by splitting it into a base language and a reactive language (Chapter 2); and a type system that restricts recursion such that only bounded space can be consumed (Chapter 3). Our contributions toward semantics are: a continuous-time-based denotational semantics for H-FRP; and the identification of a set of sufficient conditions that guarantees the convergence of H-FRP programs (Chapter 4). Our main contribution to implementation is a compiler for E-FRP that has been proved to be correct (Chapter 5). Finally, we also give some examples which illustrate the expressiveness of the RT-FRP family of languages (Chapter 3, Chapter 4, and Chapter 5).

1.6 Notations

Here we introduce some common notations that will be used throughout this thesis. More specialized notations will be introduced as they are first used. We refer the reader to the

notation table on page ix at the beginning of this thesis for a quick reference.

We write $x \equiv y$ to denote that x and y are syntactically identical, $\mathbb{N}$ for the set of natural numbers $\{1, 2, \cdots\}$, $\mathbb{N}_n$ for the set of first n natural numbers $\{1, 2, \cdots, n\}$, $\mathbb{R}$ for the set of real numbers, and $\mathbb{T}$ for the set of times (non-negative real numbers).

The set comprehension notation $\{f_j\}^{j \in S}$ is used for the set of formulas f_j where j is drawn from the set S. For example, $\{f_j\}^{j \in \mathbb{N}_5}$ means $\{f_1, f_2, f_3, f_4, f_5\}$. When S is obvious from context or uninteresting, we simply write $\{f_j\}$ instead.

Similarly, the sequence comprehension notation $\langle f_j \rangle^{j \in S}$ is used for the sequence of formulas f_j where j is drawn from the sequence S. For example, $\langle f_j \rangle^{j \in \langle 2,3,1 \rangle}$ means the sequence $\langle f_2, f_3, f_1 \rangle$. When S is obvious from context or uninteresting, we simply write $\langle f_j \rangle$ instead. The difference between a set and a sequence is that the order of elements in a sequence is significant.

Finally, we adopt the following convention on the use of fonts: the Sans-Serif font is used for identifiers in programs, the italic font is used for $meta\text{-}variables$ (i.e. variables that stand for program constructs themselves), and the bold font is for program **keywords**.

1.7 Advice to the reader

The remainder of this thesis is organized as follows: Chapter 2 presents an expression language BL that will be used by all variants of RT-FRP to express computations unrelated to time. Chapter 3 presents the RT-FRP language, and establishes its resource bound property. Then Chapter 4 introduces H-FRP, gives a continuous-time semantics to H-FRP, and proves that under suitable conditions its operational semantics converges to its continuous-time semantics as the sampling interval drops to zero. In Chapter 5 we present E-FRP, as well as a provably correct compiler for it. Finally, Chapter 6 discusses future and related work.

The reader is reminded that this thesis contains a lot of theorems and proofs. Casual or first-time readers may choose to skip the proofs, and thus the bulk of them are relegated

to Appendix A.

Chapter summary

RT-FRP is a language for real-time reactive systems, where the program has to respond to each stimulus using bounded space and time. With a carefully designed syntax and type system, RT-FRP programs are guaranteed to be resource-bounded. This thesis presents RT-FRP and its two variants (H-FRP and E-FRP) for hybrid systems and event-driven systems. We establish the convergence property of H-FRP, and prove the soundness of the compiler for E-FRP.

Chapter 2

A base language

The main entities of interest in a reactive system are behaviors (values that change over time) and events (values that are only present at discrete points in time), both having a time dimension. Therefore for RT-FRP to be a language for reactive systems, it must have features to deal with time.

At the same time, to express interesting applications, RT-FRP must also be able to perform non-trivial normal computations that do not involve time, which we call *static computations*. For example, to program a reactive logic system, the programming language should provide Boolean operations, whereas for a reactive digital control system, the language should be able to do arithmetic. As these two examples reveal, different applications often require different kinds of static computations, and hence call for different language features.

To make RT-FRP a general language for reactive systems, one approach is to make it so big that all interesting applications can be expressed in it. This, however, is unappealing to us because we believe that a single language, no matter how good, cannot apply to all applications. This is the reason why we need domain-specific languages.

The reader might wonder why we cannot use a sophisticated enough language plus customized libraries to accommodate for the differences between application domains. Our answer is two-fold:

1. all libraries need to interface with the "universal" language, and therefore have to share one type system, which can be too limiting; and

2. sometimes what matters is not what is in a language, but what is *not* in it: a "universal" language might have some features that we do not want the user to use, but it is not easy to hide language features.

Therefore, our approach is to split the RT-FRP into two parts: a *reactive language* that deals solely with time-related operations, and a *base language* that provides the ability to write static computations. In this thesis we will focus on the design and properties of the reactive language, while the user of RT-FRP can design or choose his own base language, given that certain requirements are met.

To simplify the presentation, we will work with a concrete base language we call BL (for *Base Language*). Our design goal for BL is to have a simple resource-bounded language that is just expressive enough to encode all the examples we will need in this thesis. Therefore we push for simplicity rather than power. As a result, BL is a pure, strict, first-order, and simply-typed language.

The reader should remember that BL is just *one example* of what a base language can be, and the user of RT-FRP is free to choose the base language most suitable for his particular application domain, as long as the base language satisfies certain properties. To avoid confusion, we will always write "the base language" for the unspecified base language the user might choose, and write "BL" for the particular base language we choose to use in this thesis.

2.1 Syntax

Figure 2.1 presents the abstract syntax of BL, where we use r, x, and f for the syntactic categories of real numbers, variables, and primitive functions, respectively.

An expression e can be a $()$ (read "unit"), a real number r, an n-tuple of the form

Expressions	$E \ni e$	$::=$	$() \mid r \mid (e_1, \cdots, e_n) \mid$ Nothing $\mid$ Just $e \mid$
			$f\ e \mid$ **let** $x = e$ **in** $e' \mid x \mid$
			case e **of** Just $x \Rightarrow e_1$ **else** $e_2 \mid$
			case e **of** $(x_1, \cdots, x_n) \Rightarrow e'$
Values	$V \ni v$	$::=$	$() \mid r \mid (v_1, \cdots, v_n) \mid$ Nothing $\mid$ Just v

Figure 2.1: Syntax of BL

$(e_1, \cdots, e_n)$ (where $n \geq 2$), an "optional" expression (Nothing or Just e), a function application $f\ e$, a local definition (**let-in**), a variable x, or a pattern-matching expression (**case-of**).

Nothing and Just e are the two forms of "optional" values: Nothing means a value is not present, while Just e means that one is. Optional values will play a critical role in representing events.

The **let** $x = e$ **in** e' expression defines a local variable x that can be used in e'. This construct is *not* a recursive binder, meaning that the binding of x to e is not effective in e itself. The reason for such a design is to ensure that a BL expression can always be evaluated in bounded space and time.

The **case-of** expressions have two forms: the first deconstructs an optional expression, and the second deconstructs an n-tuple.

BL is a first-order language. Therefore function arguments and return values cannot be functions themselves. Since functions cannot be returned, function applications have the form $f\ e$, where f is a function constant, rather than $e_1\ e_2$. In this thesis, we do not fully specify what f can be. Instead, we just give some examples. The reason for doing this is that the choice of functions does not affect our result for the RT-FRP language, as long as the functions being used hold certain properties. For example, they should be pure (that is, they generate the same answer given the same input) and resource-bounded (i.e. they consume bounded time and space to execute). Actually, different sets of functions can be used for different problem domains. This decoupling of function constants and the term

language makes BL a flexible base language, just as the decoupling of BL and the reactive language makes RT-FRP more flexible.

For the purpose of this thesis, it is sufficient to allow f to be an arithmetic operator ($+$, $-$, $\cdot$, $/$), power function (pow where $\mathsf{pow}(a, b)$ means "a raised to the power of b"), logic operator ($\neg$, $\vee$, $\wedge$), comparison between real numbers ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$), or trigonometric function (sin and cos).

In BL, functions all take exactly one argument. This is not a restriction as one might suspect, since an n-ary function can be written as a unary function that takes an n-tuple as its argument. For example, the $+$ function takes a pair (2-tuple) of real numbers. However, in many cases we prefer to use those binary functions as infix operators for the sake of conciseness. Hence we write $3 + 2$ and $x_1 \vee x_2$ instead of $+\,(3, 2)$ and $\vee\,(x_1, x_2)$, although the reader should remember that the former are just syntactic sugar for the latter.

When we are not interested in the value of a variable, we take the liberty of writing it as an underscore (_). This is called an *anonymous variable*, and cannot be referenced. Two variables can both be anonymous while still being distinct. For example,

$$\textbf{case}\ e\ \textbf{of}\ (\mathsf{x}, \_, \_) \Rightarrow \mathsf{x}$$

is considered semantically equivalent to

$$\textbf{case}\ e\ \textbf{of}\ (\mathsf{x}, \mathsf{y}, \mathsf{z}) \Rightarrow \mathsf{x},$$

which returns the first field of a 3-tuple e.

The reader might have noticed that BL has neither Boolean values nor conditional expressions. This is because they can be encoded in existing BL constructs and therefore are not necessary. To keep BL small, we represent False and True by Nothing and Just $()$

respectively, and then the conditional expression

$$\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2$$

can be expressed as

$$\textbf{case } e \textbf{ of } \mathsf{Just}\ \_ \Rightarrow e_1 \textbf{ else } e_2.$$

From the grammar for e and v, it is obvious that any value v is also an expression. In other words, $V \subseteq E$.

2.2 Type system

BL is a simply-typed first-order language, which means that there are no polymorphic functions and that all functions must be applied in full. Although the type system for BL is not very interesting, we present it here for the completeness of this thesis.

2.2.1 Type language

The type language of BL is given by Figure 2.2.

$$\text{Types} \quad \alpha \quad ::= \quad () \mid \mathsf{Real} \mid (\alpha_1, \cdots, \alpha_n) \mid \mathsf{Maybe}\ \alpha$$

Figure 2.2: Type language of BL

Note that $()$ is used in both the term language and the type language. In the term language, $()$ is the "unit" value, and in the type language, it stands for the *unit type*, whose only value is "unit," or $()$.

Real is the type for real numbers.

Similar to $()$, the tuple notation $(,,)$ is used at both the term and the type level. In the type language, $(\alpha_1, \cdots, \alpha_n)$ is the type of an n-tuple whose j-th field has type α_j.

Maybe α is the type for optional values, which can be either Nothing or Just v where v has type α.

We will use Bool as a synonym for the type Maybe $()$, whose only values are Nothing and Just $()$. As said in Section 2.1, we let False and True be our preferred way of writing Nothing and Just $()$.

Since functions are not first-class in BL, we do not need to have function types in the type language.

2.2.2 Context

The typing of BL terms is done in a *variable context* (sometimes simply called a *context*), which assigns types to variables that occur free in an expression.

The *free variables* in an expression e, written as $FV(e)$, are defined as:

$\boxed{FV(e)}$

$$FV(()) = \emptyset, \quad FV(r) = \emptyset, \quad FV((e_1, \cdots, e_n)) = \bigcup_{j=1}^{n} FV(e_j)$$

$$FV(\mathsf{Nothing}) = \emptyset, \quad FV(\mathsf{Just}\ e) = FV(e), \quad FV(f\ e) = FV(e)$$

$$FV(\textbf{let}\ x = e\ \textbf{in}\ e') = FV(e) \cup (FV(e') - \{x\}), \quad FV(x) = \{x\}$$

$$FV(\textbf{case}\ e\ \textbf{of}\ \mathsf{Just}\ x \Rightarrow e_1\ \textbf{else}\ e_2) = FV(e) \cup (FV(e_1) - \{x\}) \cup FV(e_2)$$

$$FV(\textbf{case}\ e\ \textbf{of}\ (x_1, \cdots, x_n) \Rightarrow e') = FV(e) \cup (FV(e') - \{x_1, \cdots, x_n\})$$

A context Γ has the syntax

$$\Gamma \quad ::= \quad \{x_j : \alpha_j\}$$

We require the variables in a context to be distinct. Hence,

$$\{\mathsf{x} : \mathsf{Real}, \mathsf{y} : \mathsf{Real}, \mathsf{x} : ()\}$$

is *not* a context since the variable x occurs twice. We will only consider well-formed con-

texts in this thesis.

Notations

Before presenting the type system, we define some notations we will use with contexts.

We use $\uplus$ for non-overlapping set union. Therefore, when the reader sees an expression of the form

$$A \uplus B,$$

it should be interpreted as $A \cup B$, where $A \cap B = \emptyset$.

Often we write

$$\Gamma(x) = \alpha$$

to indicate the fact that $\Gamma \equiv \Gamma' \uplus \{x : \alpha\}$ for some Γ'. Since a variable can occur at most once in Γ, it is obvious that if $\Gamma(x)$ exists, it is uniquely determined. This justifies our use of the function-like notation $\Gamma(x)$.

Given a context Γ, its *domain* is defined as the set of variables occurring in Γ, and written as $\mathrm{dom}(\Gamma)$. The formal definition of $\mathrm{dom}(\Gamma)$ is given by:

$$\mathrm{dom}(\{x_j : \alpha_j\}^{j \in J}) = \{x_j\}^{j \in J}.$$

Finally, we write

$$\Gamma \oplus \{x_j : \alpha_j\}$$

for a context identical to Γ except that x_j is assigned type α_j. The definition of $\oplus$ is given by:

$$\begin{aligned} \Gamma \oplus \Gamma' \quad = \quad & \{x : \Gamma(x) \;\big|\; x \in \mathrm{dom}(\Gamma) - \mathrm{dom}(\Gamma')\} \uplus \\ & \{x : \Gamma'(x) \;\big|\; x \in \mathrm{dom}(\Gamma')\}. \end{aligned}$$

The reader should note that $\oplus$ is *not* commutative and therefore is *not* to be confused with the set union operator $\cup$.

2.2.3 Typing judgments

We use two judgments to define the type system of BL:

- $f : \alpha \to \alpha'$: "f is a function from α to α';" and
- $\Gamma \vdash e : \alpha$: "e has type α in context Γ."

As shorthand, we write

$$f_1, \cdots, f_n : \alpha \to \alpha'$$

for that $f_1 : \alpha \to \alpha'$, ..., and $f_n : \alpha \to \alpha'$.

BL is a first-order monomorphic language, which makes its type system fairly easy to understand. Figure 2.3 presents the BL type system, which basically says:

The term $()$ has type $()$, or unit. A real number r has type Real. A tuple $(e_1, \cdots, e_n)$ has type $(\alpha_1, \cdots, \alpha_n)$ where e_j has type α_j for all $j \in \mathbb{N}_n$. Nothing and Just e has type Maybe α where e has type α. A function $f : \alpha \to \alpha'$ takes an expression e of type α and the result $f\ e$ has type α'.

Given a context Γ, **let** $x = e$ **in** e' has type α' where e has type α in Γ and e' has type α' in the context $\Gamma \oplus \{x : \alpha\}$. Note that x is not added to Γ to type e, which corresponds to the fact that **let** is not a recursive binder.

The type of a variable x is dictated by the context.

For **case** e **of** Just $x \Rightarrow e_1$ **else** e_2, we require e to have type Maybe α. Then the whole expression is typed α' if e_1 has type α' (with x assigned type α) and e_2 has type α'.

Finally, **case** e **of** $(x_1, \cdots, x_n) \Rightarrow e'$ has type α' where e has type $(\alpha_1, \cdots, \alpha_n)$ and e' has type α' (with x_j assigned type α_j for all $j \in \mathbb{N}_n$).

Generally speaking, in a functional language, a value can be represented as a *closure*, which is a data structure that holds an expression and an environment of variable bindings in which the expression is to be evaluated. In BL, a value term cannot contain free variables, and therefore there is no need to include an environment in the representation of a value. This suggests that the context is not relevant at all in the typing of a value. This

$\boxed{f : \alpha \to \alpha'}$

$$\frac{}{+, -, \cdot, /, \mathsf{pow} : (\mathsf{Real}, \mathsf{Real}) \to \mathsf{Real}}$$

$$\frac{}{\neg : \mathsf{Bool} \to \mathsf{Bool}} \qquad \frac{}{\vee, \wedge : (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}}$$

$$\frac{}{=, \neq, <, \leq, >, \geq : (\mathsf{Real}, \mathsf{Real}) \to \mathsf{Bool}} \qquad \frac{}{\mathsf{sin}, \mathsf{cos} : \mathsf{Real} \to \mathsf{Real}}$$

$\boxed{\Gamma \vdash e : \alpha}$

$$\frac{}{\Gamma \vdash () : ()} \qquad \frac{}{\Gamma \vdash r : \mathsf{Real}} \qquad \frac{\{\Gamma \vdash e_j : \alpha_j\}^{j \in \mathbb{N}_n}}{\Gamma \vdash (e_1, \cdots, e_n) : (\alpha_1, \cdots, \alpha_n)} \, (n \geq 2)$$

$$\frac{}{\Gamma \vdash \mathsf{Nothing} : \mathsf{Maybe}\ \alpha} \qquad \frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \mathsf{Just}\ e : \mathsf{Maybe}\ \alpha} \qquad \frac{f : \alpha \to \alpha' \quad \Gamma \vdash e : \alpha}{\Gamma \vdash f\ e : \alpha'}$$

$$\frac{\Gamma \vdash e : \alpha \quad \Gamma \oplus \{x : \alpha\} \vdash e' : \alpha'}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \alpha'} \qquad \frac{}{\Gamma \uplus \{x : \alpha\} \vdash x : \alpha}$$

$$\frac{\Gamma \vdash e : \mathsf{Maybe}\ \alpha \quad \Gamma \oplus \{x : \alpha\} \vdash e_1 : \alpha' \quad \Gamma \vdash e_2 : \alpha'}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \mathsf{Just}\ x \Rightarrow e_1\ \textbf{else}\ e_2 : \alpha'}$$

$$\frac{\Gamma \vdash e : (\alpha_1, \cdots, \alpha_n) \quad \Gamma \oplus \{x_j : \alpha_j\}^{j \in \mathbb{N}_n} \vdash e' : \alpha'}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ (x_1, \cdots, x_n) \Rightarrow e' : \alpha'} \, (x_1, \cdots, x_n \text{ are distinct})$$

Figure 2.3: Type system of BL

intuition is formalized by the following lemma:

Lemma 2.1 (Typing of values). $\Gamma \vdash v : \alpha \iff \emptyset \vdash v : \alpha$.

Proof. The proof for the forward direction is by induction on the derivation of $\Gamma \vdash v : \alpha$, and the backward direction is proved by induction on the derivation of $\emptyset \vdash v : \alpha$. □

The decision of keeping values from containing free variables helps to ensure that a value can always be stored in bounded space, which is not the case in general if values can contain free variables. Also, having no free variables in values makes it easier to ensure that any RT-FRP computation can be done in bounded time.

Lemma 2.1 shows that the type of a value v has nothing to do with the context in which it is typed, and hence is an intrinsic property of v itself. Therefore, we can write

$$v \in \alpha$$

to indicate the fact that $\Gamma \vdash v : \alpha$ for some Γ. With this notation, we can treat a type α as the set of all values of that type, i.e. $\{v \mid \emptyset \vdash v : \alpha\}$.

2.3 Operational semantics

Given an expression e and the values of the free variables in e, we expect to be able to "evaluate" e to some value v. The *semantics* of BL defines how we can find such a v. In this section, we give a big-step operational semantics for BL, which is presented as a ternary relation between e, the binding of free variables in e, and v.

We call the binding of values to variables an *environment*. The evaluation of an expression is always done in some particular environment. The syntax of an environment $\mathcal{X}$ is defined as

$$\mathbb{X} \ni \mathcal{X} \quad ::= \quad \{x_j \mapsto v_j\}.$$

To distinguish from the *signal environment* that will be introduced in Chapter 3, we sometimes also call $\mathcal{X}$ a *variable environment*.

We require variables defined in an environment to be distinct, hence justifying the use of the notation

$$\mathcal{X}(x) = v$$

for the fact that $\mathcal{X} \equiv \mathcal{X}' \uplus \{x \mapsto v\}$ for some $\mathcal{X}'$.

Like what we do for contexts, we write

$$\mathcal{X} \oplus \{x_j \mapsto v_j\}$$

for an environment identical to $\mathcal{X}$ except that x_j maps to v_j.

Often, we will need to assert that the variables are assigned values of the right types in an environment, for which the concept of *environment-context compatibility* is invented:

Definition 2.1 (Environment-context compatibility). We say that $\mathcal{X} \equiv \{x_j \mapsto v_j\}^{j \in J}$ is *compatible* with $\Gamma \equiv \{x_j : \alpha_j\}^{j \in J}$, if for all $j \in J$ we have $\Gamma \vdash v_j : \alpha_j$. We write this as $\Gamma \vdash \mathcal{X}$.

As we said, the evaluation relation is between an expression e, an environment $\mathcal{X}$, and a value v. We use two evaluation judgments to define this relation:

- $f\ v \hookrightarrow v'$: "applying function f to value v yields v';" and
- $\mathcal{X} \vdash e \hookrightarrow v$: "$e$ evaluates to v in environment $\mathcal{X}$."

The functions and operators of BL have their standard semantics (for example, we expect $3 + 2 \hookrightarrow 5$), and we hence omit the definition of the judgment $f\ v \hookrightarrow v'$. Figure 2.4 defines $\mathcal{X} \vdash e \hookrightarrow v$.

2.4 Properties of BL

When designing the BL language, we expect it to have certain properties, such as:

$$\boxed{\mathcal{X} \vdash e \hookrightarrow v}$$

$$\frac{}{\mathcal{X} \vdash () \hookrightarrow ()} \qquad \frac{}{\mathcal{X} \vdash r \hookrightarrow r} \qquad \frac{\{\mathcal{X} \vdash e_j \hookrightarrow v_j\}^{j \in \mathbb{N}_n}}{\mathcal{X} \vdash (e_1, \cdots, e_n) \hookrightarrow (v_1, \cdots, v_n)}$$

$$\frac{}{\mathcal{X} \vdash \mathsf{Nothing} \hookrightarrow \mathsf{Nothing}} \qquad \frac{\mathcal{X} \vdash e \hookrightarrow v}{\mathcal{X} \vdash \mathsf{Just}\ e \hookrightarrow \mathsf{Just}\ v}$$

$$\frac{\mathcal{X} \vdash e \hookrightarrow v \quad f\ v \hookrightarrow v'}{\mathcal{X} \vdash f\ e \hookrightarrow v'}$$

$$\frac{\mathcal{X} \vdash e \hookrightarrow v \quad \mathcal{X} \oplus \{x \mapsto v\} \vdash e' \hookrightarrow v'}{\mathcal{X} \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' \hookrightarrow v'} \qquad \frac{}{\mathcal{X} \uplus \{x \mapsto v\} \vdash x \hookrightarrow v}$$

$$\frac{\mathcal{X} \vdash e \hookrightarrow \mathsf{Just}\ v \quad \mathcal{X} \oplus \{x \mapsto v\} \vdash e_1 \hookrightarrow v'}{\mathcal{X} \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathsf{Just}\ x \Rightarrow e_1\ \mathbf{else}\ e_2 \hookrightarrow v'}$$

$$\frac{\mathcal{X} \vdash e \hookrightarrow \mathsf{Nothing} \quad \mathcal{X} \vdash e_2 \hookrightarrow v}{\mathcal{X} \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathsf{Just}\ x \Rightarrow e_1\ \mathbf{else}\ e_2 \hookrightarrow v}$$

$$\frac{\mathcal{X} \vdash e \hookrightarrow (v_1, \cdots, v_n) \quad \mathcal{X} \oplus \{x_j \mapsto v_j\}^{j \in \mathbb{N}_n} \vdash e' \hookrightarrow v}{\mathcal{X} \vdash \mathbf{case}\ e\ \mathbf{of}\ (x_1, \cdots, x_n) \Rightarrow e' \hookrightarrow v}$$

Figure 2.4: Operational semantics of BL

1. evaluating a well-typed expression should always lead to some value — the evaluation process will not get stuck;

2. the evaluation relation should be deterministic, i.e. given $\mathcal{X}$, if e evaluates to v in $\mathcal{X}$, then v is uniquely determined;

3. evaluation should preserve type: whenever e evaluates to v, e and v have the same type; and

4. the evaluation of an expression can be done in bounded number of derivation steps.

In the rest of the section, we will establish these properties of BL. In Chapter 3, we will use them to prove corresponding properties of RT-FRP. Uninterested readers can skip to the next section now, and return later to reference these properties as the need arises.

First, we should give firm definitions to the properties we want to study:

Definition 2.2 (Properties of BL). We say that

1. BL is *productive*, if $(\Gamma \vdash e : \alpha \text{ and } \Gamma \vdash \mathcal{X}) \implies \exists v.\ \mathcal{X} \vdash e \hookrightarrow v$;

2. BL is *deterministic*, if $(\mathcal{X} \vdash e \hookrightarrow v \text{ and } \mathcal{X} \vdash e \hookrightarrow v') \implies v \equiv v'$;

3. BL is *type-preserving*, if $(\Gamma \vdash e : \alpha, \Gamma \vdash \mathcal{X}, \text{ and } \mathcal{X} \vdash e \hookrightarrow v) \implies \Gamma \vdash v : \alpha$; and

4. BL is *resource-bounded*, if given $\Gamma \vdash e : \alpha$, $\Gamma \vdash \mathcal{X}$, and $\mathcal{X} \vdash e \hookrightarrow v$, the size of the derivation tree for $\mathcal{X} \vdash e \hookrightarrow v$ is bounded by a value that depends only on e.

As BL is parameterized by a set of built-in functions, in order to establish these properties we need to define similar properties for functions:

Definition 2.3 (Properties of functions). Given a function $f : \alpha \to \alpha'$, we say that:

1. f is *productive*, if for all $v \in \alpha$, there is a v' such that $f\ v \hookrightarrow v'$;

2. f is *deterministic*, if for all v, $(f\ v \hookrightarrow v' \text{ and } f\ v \hookrightarrow v'') \implies v' \equiv v''$;

3. f is *type-disciplined*, if for all $v \in \alpha$, $f\ v \hookrightarrow v'$ implies that $v' \in \alpha'$; and

4. f is *resource-bounded*, if for all $v \in \alpha$, $f\ v \hookrightarrow v' \implies$ the size of the derivation tree for $f\ v \hookrightarrow v'$ is bounded by a number that depends only on f and v.

Now we can proceed to establish the properties of BL by a series of lemmas. First, we repeat Lemma 2.1 here:

Lemma 2.1 (Typing of values). $\Gamma \vdash v : \alpha \iff \emptyset \vdash v : \alpha$.

Next, we show that the evaluation of a well-typed expression will not get stuck:

Lemma 2.2 (Productivity). BL is productive if all functions are productive.

Proof. First, we need to prove the following proposition:

"If $\Gamma \vdash \mathcal{X}$ and $\forall j \in J.\ \Gamma \vdash v_j : \alpha_j$, then $\Gamma \oplus \{x_j : \alpha_j\}^{j \in J} \vdash \mathcal{X} \oplus \{x_j : v_j\}^{j \in J}$."

The proof is by the definition of $\oplus$ and Lemma 2.1.

The rest of the proof for this lemma is by induction on the derivation of $\Gamma \vdash e : \alpha$, and we will be using Lemma 2.4 which we will soon prove. □

Given an expression, we expect that its value only depends on the free variables in it. This intuition can be presented formally as:

Lemma 2.3 (Relevant variables). If all functions are deterministic, $\mathcal{X} \vdash e \hookrightarrow v$, $\mathcal{X}' \vdash e \hookrightarrow v'$, and $\mathcal{X}(x) \equiv \mathcal{X}'(x)$ for all $x \in FV(e)$, then we have $v \equiv v'$.

Proof. By induction on the derivation of $\mathcal{X} \vdash e \hookrightarrow v$. □

By directly applying Lemma 2.3, we get the following corollary:

Corollary 2.1 (Determinism). BL is deterministic if all functions are deterministic.

The next lemma assures us of the type of the return value when evaluating an expression:

Lemma 2.4 (Type preservation). BL is type-preserving if all functions are type-disciplined.

Proof. The proof is by induction on the derivation tree of $\Gamma \vdash e : \alpha$. □

We are most interested in this property of BL:

Lemma 2.5 (Resource bound). BL is resource-bounded if all functions are resource bounded.

Proof. For any e and $\mathcal{X}$ where $\Gamma \vdash e : \alpha$ and $\Gamma \vdash \mathcal{X}$, the proof is by induction on the derivation tree of $\Gamma \vdash e : \alpha$. □

A BL value cannot contain free variables:

Lemma 2.6 (Free variables in values). For any given v, $FV(v) = \emptyset$.

Proof. By induction on the structure of v. □

Obviously, a value evaluates to itself in any environment:

Lemma 2.7 (Evaluation of value). For any given $\mathcal{X}$ and v, $\mathcal{X} \vdash v \hookrightarrow v$.

Proof. By induction on the structure of v. □

If we replace a free variable by its value in an expression e, the value of e should remain the same:

Lemma 2.8 (Substitution). If $\{x \mapsto v\} \uplus \mathcal{X} \vdash e \hookrightarrow v'$ and $\mathcal{X} \vdash e[x := v] \hookrightarrow v''$, then $v' \equiv v''$.

Proof. By induction on the derivation of $\{x \mapsto v\} \uplus \mathcal{X} \vdash e \hookrightarrow v'$. The critical case is when $e \equiv x$, where we need to show that $\{x \mapsto v\} \uplus \mathcal{X} \vdash x \hookrightarrow v$ and $\mathcal{X} \vdash v \hookrightarrow v$. The former is by the definition of the operational semantics, and the latter is by direct application of Lemma 2.7. □

Furthermore, we can show that a value of type Maybe α must be in one of the two forms Nothing and Just v:

Lemma 2.9 (Forms of optional values). $\Gamma \vdash v :$ Maybe α implies that either $v \equiv$ Nothing or $v \equiv$ Just v' for some v' where $\Gamma \vdash v' : \alpha$.

Proof. By inspecting the typing rules in Figure 2.3. □

Chapter summary

A reactive programming language needs to deal with both time-related computations and static computations. We hence split RT-FRP into two orthogonal parts: a reactive language and a base language. While the reactive language is fixed, the user can choose the base language for use with his particular application.

This chapter presents one instance of base languages called BL. BL is a pure, strict, first-order, and simply-typed language. A type system and an operational semantics are given to BL. As preparation for establishing the resource-boundedness of RT-FRP, certain properties of BL are also explored.

Chapter 3

Real-Time FRP

The Real-Time FRP (RT-FRP) language is composed of two parts: a base language for expressing static computations and a reactive language that deals with time and reaction. We have given an instance of base languages in the previous chapter. This chapter gives a thorough treatment to the reactive language.

In addition to giving the syntax and type system of RT-FRP, we prove the resource-boundedness of this language, which has been one of our primary design goals. We also provide evidence that RT-FRP is practical by solving a variety of problems typical in real-time reactive systems.

RT-FRP is deliberately small such that its properties can be well studied. For example, it lacks mutually recursive signal definitions and pattern matching. Yet this language is expressive enough that many of the missing features can be defined as syntactic sugar.

This chapter is organized as follows: Section 3.1 defines the syntax of the reactive part of RT-FRP, and informally explains each construct. Section 3.2 presents an operational semantics for untyped RT-FRP programs and the mechanics of executing an RT-FRP program. Section 3.3 discusses the difference between the two kinds of recursion RT-FRP supports. Section 3.4 explains what can go wrong with an untyped program, and solves the problem by restricting the programs with a type system. Well-typed programs are shown to run in bounded space and time in Section 3.5, where we also establish the space

and time complexity of executing a program. Section 3.6 discusses the expressiveness of RT-FRP, using the example of a cruise control system. Finally, interested readers can find some syntactic sugar for RT-FRP in Section 3.7 that makes it easier to use.

3.1 Syntax

To get the reader started with RT-FRP, we first present its syntax. The base language syntax (terms e and v) has been defined in the previous chapter. The reactive part of RT-FRP is given by Figure 3.1, where z is the syntactic category of *signal variables*. Note that base language terms e can occur inside signal terms s, but not the other way around. Furthermore, a variable bound by **let-snapshot** can only be used in the base language.

$$
\begin{array}{lrcl}
\text{Signals} & S \ni s & ::= & \textbf{input} \mid \textbf{time} \mid \textbf{let snapshot}\ x \leftarrow s_1\ \textbf{in}\ s_2 \mid \textbf{ext}\ e \mid \\
 & & & \textbf{delay}\ e\ s \mid \textbf{let signal}\ \{\, z_j(x_j) = u_j \,\}\ \textbf{in}\ s \mid u \\
\text{Switchers} & u & ::= & s\ \textbf{until}\ \langle\, s_j \ \Rightarrow\ z_j \,\rangle
\end{array}
$$

Figure 3.1: Syntax of RT-FRP

The syntactic category u is for a special kind of signals called "switchers." The reader might question the necessity for a separate syntactic category for switchers. Indeed, we can get rid of u and define s as:

$$
\begin{array}{rcl}
s & ::= & \textbf{input} \mid \textbf{time} \mid \textbf{let snapshot}\ x \leftarrow s_1\ \textbf{in}\ s_2 \mid \textbf{ext}\ e \mid \\
 & & \textbf{delay}\ e\ s \mid \textbf{let signal}\ \{\, z_j(x_j) = s_j \,\}\ \textbf{in}\ s \mid s\ \textbf{until}\ \langle\, s_j \ \Rightarrow\ z_j \,\rangle
\end{array}
$$

However, this syntax accepts more programs that might grow arbitrarily large, and therefore requires a more complicated type system to ensure resource bounds. This is the reason why we choose the current syntax.

Those familiar with FRP may have noticed that unlike FRP, RT-FRP does not have

syntactic distinction between behaviors (values that change over time) and events (values that are present only at discrete points in time). Instead, they are unified under the syntactic category *signals*. A signal that carries values of a "Maybe α" type is treated as an event, and the rest of the signals are behaviors. We say that an event is *occurring* if its current value is "Just something", or that it is *not occurring* if its current value is "Nothing." When we know a signal is an event, we often name it as ev to emphasize the fact.

To simplify the presentation of the operational semantics, but without loss of generality, we require that *all signal variables (z) in a program have distinct names*.

In the rest of this section we explain each of the reactive constructs of RT-FRP in more detail.

Primitive signals

The two primitive signals in RT-FRP are **input**, the current stimulus, and **time**, the current time in seconds. In this paper we only make use of **time**, as it is sufficient for illustrating all the operations that one might want to define on external stimuli. In practice, **input** will carry values of different types – such as mouse clicks, keyboard presses, network messages, and so on – for different applications, since there are few interesting systems that react only to **time**.

Interfacing with the base language

The reactive part of RT-FRP does not provide primitive operations such as addition and subtraction on the values of signals. Instead, this is relegated to the base language. To interface with the base language, the reactive part of RT-FRP has a mechanism for exporting snapshots of signal values to the base language, and a mechanism for importing base language values back into the signal world. Specifically, to export a signal, we snapshot its current value using the **let-snapshot** construct, and to invoke an external computation in the base language we use the **ext** e construct.

To illustrate, suppose we wish to define a signal representing the current time in minutes. We can do this by:

$$\textbf{let snapshot}\ \mathsf{t} \leftarrow \textbf{time in ext}\ (\mathsf{t}/60)$$

To compute externally with more than one signal, we have to snapshot each one separately. For example, the term:

$$\begin{array}{l}\textbf{let snapshot}\ \mathsf{x} \leftarrow s_1\ \textbf{in} \\ \quad \textbf{let snapshot}\ \mathsf{y} \leftarrow s_2\ \textbf{in ext}\ (\mathsf{x}+\mathsf{y})\end{array}$$

is a signal that is the pointwise sum of the signals s_1 and s_2.

Accessing previous values

The signal **delay** $e\ s$ is a delayed version of s, whose initial value is the value of e.

To illustrate the use of **delay**, the following term computes the difference between the current time and the time at the previous program execution step:

$$\begin{array}{l}\textbf{let snapshot}\ \mathsf{t0} \leftarrow \textbf{time in} \\ \quad \textbf{let snapshot}\ \mathsf{t1} \leftarrow \textbf{delay}\ 0\ \textbf{time in ext}\ (\mathsf{t0}-\mathsf{t1})\end{array}$$

Switching

As in FRP, a signal can react to an event by switching to another signal. The **let-signal** construct defines a group of signals (or more precisely *signal templates*, as they are each parameterized by a variable). Each of the signals must be a switcher (an **until** construct), which has the syntax "s **until** $\langle s_j \Rightarrow z_j \rangle$" and is a signal s that switches to another signal z_j when some event s_j occurs. For example,

$$s\ \textbf{until}\ \langle s_1 \Rightarrow z_1,\ s_2 \Rightarrow z_2 \rangle$$

initially behaves as s, and becomes $z_1(v)$ if event s_1 occurs with value v, or becomes $z_2(v)$ if s_1 does not occur but event s_2 occurs with value v. Please note that a switcher switches only once and then ignores the events. Section 3.7.5 shows how repeated switching can be done in RT-FRP.

Signal templates introduced by the same **let-signal** construct can be mutually recursive, i.e. they can switch into each other. However, as we will see in Section 3.4.1, unrestricted switching leads to unbounded space consumption. Hence when defining the type system of RT-FRP in Section 3.4, we will impose subtle restriction on recursive switching to make sure it is well-behaved.

As an example of switching, a sample-and-hold register that remembers the most recent event value it has received can be defined as:

$$\textbf{let signal } \mathsf{hold}(\mathsf{x}) = (\textbf{ext } \mathsf{x}) \textbf{ until } sample \Rightarrow \mathsf{hold}$$
$$\textbf{in } \underline{(\textbf{ext } 0) \textbf{ until } sample \Rightarrow \mathsf{hold}}$$

This signal starts out as 0. Whenever the event $sample$ occurs, its current value is substituted for x in the body of $\mathsf{hold}(\mathsf{x})$, and that value becomes the current value of the overall signal. As we will see in Section 3.7.7, the underlined part can be simplified to $\mathsf{hold}(0)$ using syntactic sugar.

A switcher u may have arbitrary number of switching branches, including zero. When there is no switching branch, u has the form "s **until** $\langle\rangle$," and behaves just like s since there is no event to trigger it to switch. Indeed, as we will see in Section 3.2, "s **until** $\langle\rangle$" and "s" have the same operational semantics.

It is worth pointing out though that "s **until** $\langle\rangle$" is a switcher while "s" is not. Since the definition bodies in a "**let-signal**" must be switchers, the programmer must use "s **until** $\langle\rangle$" instead of "s" there. After seeing the type system in Section 3.4, the reader will discover that "s **until** $\langle\rangle$" and "s" follow different typing rules, which is necessary to ensure the resource-boundedness of RT-FRP.

3.2 Operational semantics

Now that we have explained what each construct in RT-FRP does, the reader should have some intuition about how to interpret an arbitrary RT-FRP program. To transform the intuition into precise understanding, we present the full details of the operational semantics of RT-FRP in this section.

3.2.1 Environments

Program execution takes place in the context of a *variable environment* $\mathcal{X}$ (as defined in Section 2.3) and a *signal environment* $\mathcal{Z}$:

$$\mathcal{Z} \quad ::= \quad \{z_j \mapsto \lambda x_j.u_j\}.$$

The variable environment $\mathcal{X}$ is used to store the current values of signals, and hence maps signal variables to values. The signal environment $\mathcal{Z}$ maps a signal variable to its definition. The lambda abstraction makes explicit the formal argument and the signal parameterized by that argument.

3.2.2 Two judgment forms

We define the single-step semantics by means of two judgments:

- $\mathcal{X} \vdash s \overset{t,i}{\hookrightarrow} v$: "$s$ evaluates to v on input i at time t, in environment $\mathcal{X}$;" and
- $\mathcal{X}; \mathcal{Z} \vdash s \overset{t,i}{\longrightarrow} s'$: "$s$ is updated to become s' on input i at time t, in environments $\mathcal{X}$ and $\mathcal{Z}$."

Sometimes we combine the two judgments and write

$$\mathcal{X}; \mathcal{Z} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v, s'.$$

When the environments are empty, we simplifies the notation further to

$$s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v, s'.$$

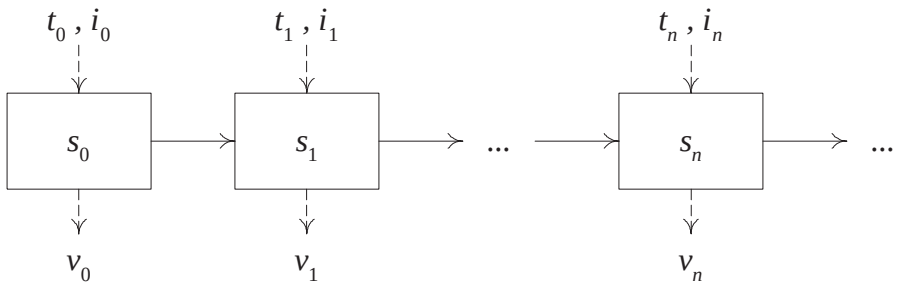
Note that the evaluation of a term s does not depend on the signal environment, and the semantics for each step is parameterized by the current time t and the current input i.

The role of evaluation is to compute the output of a term. The role of updating is to modify the state of the program. We present the rules of each of these judgments in Figure 3.2 and 3.3.

The overall execution, or "run", of an RT-FRP program is modeled as an infinite sequence of interactions with the environment, and in that sense does not terminate. Formally, for a given sequence of time-stamped stimuli $\langle (t_0, i_0), (t_1, i_1), \dots \rangle$, a *run* of an RT-FRP program s_0 produces a sequence of values $\langle v_0, v_1, \dots \rangle$, where $s_k \overset{t_k,i_k}{\hookrightarrow\!\!\!\!\rightarrow} v_k, s_{k+1}$ for all $k \in \{0\} \cup \mathbb{N}$. Thus, a run can be visualized as an infinite chain of the form:

$$\mathcal{X};\ \mathcal{Z} \vdash s_0 \overset{t_0,i_0}{\hookrightarrow\!\!\!\!\rightarrow} v_0, s_1 \overset{t_1,i_1}{\hookrightarrow\!\!\!\!\rightarrow} v_1, s_2 \dots s_n \overset{t_n,i_n}{\hookrightarrow\!\!\!\!\rightarrow} v_n, s_{n+1} \dots$$

or as in the following figure:



where we use "$\rightarrow$" for update of terms, and "$\dashrightarrow$" for flow of data.

Next, we explain the judgment rules in detail.

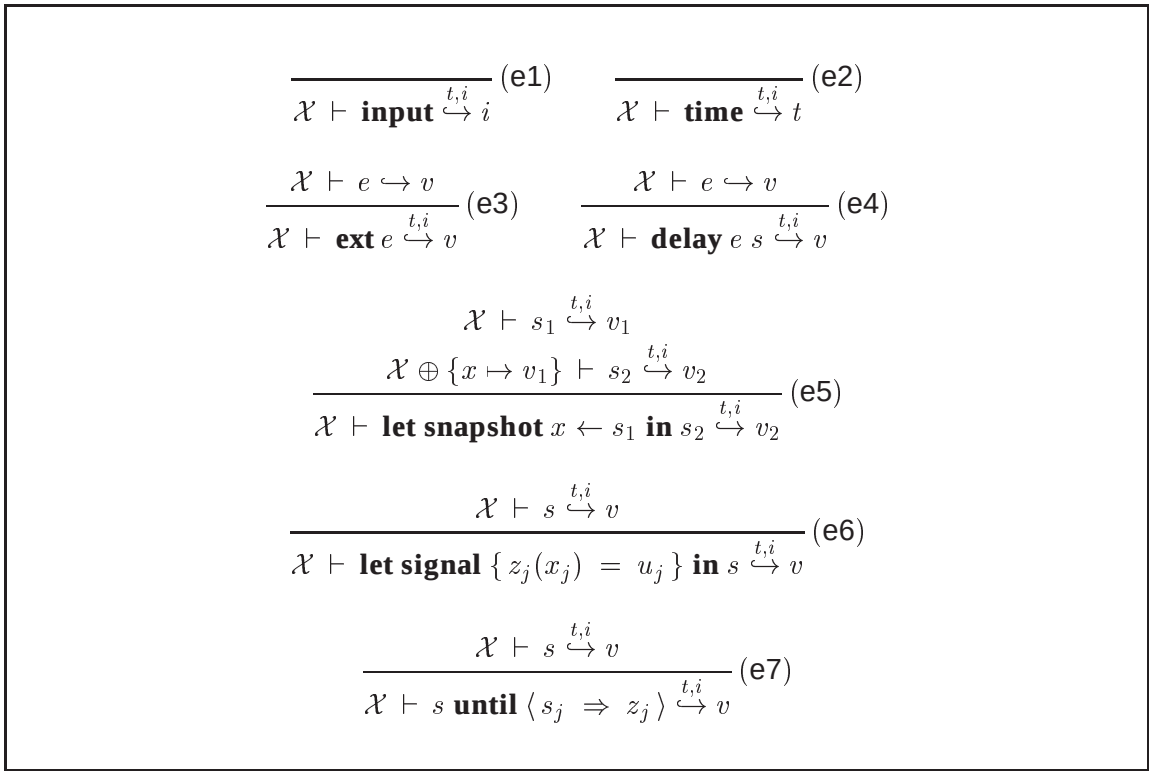


Figure 3.2: Operational semantics for RT-FRP: evaluation rules

3.2.3 The mechanics of evaluating and updating

Evaluating the **input** and **time** constructs simply returns the current value for the input and the time, respectively (rules e1 and e2). Updating these two constructs leaves them unchanged (rules u1 and u2).

The evaluation rule for calls to the base language (rule e3) uses the judgment $\mathcal{X} \vdash e \hookrightarrow v$ to evaluate the BL term e, which was defined in Section 2.3. The term **ext** e is not changed by updating (rule u3).

The instantaneous value of **delay** $e\ s$ is just the value of e (rule e4). But it is updated to a new term **delay** $v'\ s'$, where v' is the previous instantaneous value of s, and s' is the new term resulting from updating s (rule u4).

Evaluation of **let snapshot** $x \leftarrow s_1$ **in** s_2 consists of two stages: first, s_1 is evaluated to get v; second, s_2 is evaluated with x bound to v, yielding the result (rule e5). Updating the term is done by updating both s_1 and s_2 (rule u5).

$$\frac{}{\mathcal{X};\ \mathcal{Z}\ \vdash\ \mathbf{input} \xrightarrow{t,i} \mathbf{input}}\ (\text{u1}) \qquad \frac{}{\mathcal{X};\ \mathcal{Z}\ \vdash\ \mathbf{time} \xrightarrow{t,i} \mathbf{time}}\ (\text{u2})$$

$$\frac{}{\mathcal{X};\ \mathcal{Z}\ \vdash\ \mathbf{ext}\ e \xrightarrow{t,i} \mathbf{ext}\ e}\ (\text{u3}) \qquad \frac{\mathcal{X};\ \mathcal{Z}\ \vdash\ s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v', s'}{\mathcal{X};\ \mathcal{Z}\ \vdash\ \mathbf{delay}\ e\ s \xrightarrow{t,i} \mathbf{delay}\ v'\ s'}\ (\text{u4})$$

$$\frac{\begin{array}{c}\mathcal{X}\ \vdash\ s_1 \overset{t,i}{\hookrightarrow} v_1 \\ \mathcal{X} \oplus \{x \mapsto v_1\}; \mathcal{Z}\ \vdash\ s_1 \xrightarrow{t,i} s_1' \quad \mathcal{X} \oplus \{x \mapsto v_1\}; \mathcal{Z}\ \vdash\ s_2 \xrightarrow{t,i} s_2'\end{array}}{\mathcal{X};\ \mathcal{Z}\ \vdash\ \mathbf{let\ snapshot}\ x \leftarrow s_1\ \mathbf{in}\ s_2 \xrightarrow{t,i} \mathbf{let\ snapshot}\ x \leftarrow s_1'\ \mathbf{in}\ s_2'}\ (\text{u5})$$

$$\frac{\mathcal{X};\ \mathcal{Z} \oplus \{z_j \mapsto \lambda x_j.u_j\}\ \vdash\ s \xrightarrow{t,i} s'}{\mathcal{X};\ \mathcal{Z}\ \vdash\ \mathbf{let\ signal}\ \{\, z_j(x_j)\ =\ u_j\,\}\ \mathbf{in}\ s \xrightarrow{t,i} \mathbf{let\ signal}\ \{\, z_j(x_j)\ =\ u_j\,\}\ \mathbf{in}\ s'}\ (\text{u6})$$

$$\frac{\{\, \mathcal{X}\ \vdash\ s_j \overset{t,i}{\hookrightarrow} \mathsf{Nothing}\,\}^{j \in \mathbb{N}_{m-1}} \qquad \mathcal{X}\ \vdash\ s_m \overset{t,i}{\hookrightarrow} \mathsf{Just}\ v}{\mathcal{X};\ \mathcal{Z} \uplus \{z_m \mapsto \lambda x.u\}\ \vdash s\ \mathbf{until}\ \langle\, s_j\ \Rightarrow\ z_j\,\rangle \xrightarrow{t,i} u[x := v]}\ (\text{u7})$$

$$\frac{\mathcal{X};\emptyset\ \vdash\ s \xrightarrow{t,i} s' \qquad \{\, \mathcal{X};\emptyset\ \vdash\ s_j \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} \mathsf{Nothing}, s_j'\,\}}{\mathcal{X};\ \mathcal{Z}\ \vdash\ s\ \mathbf{until}\ \langle\, s_j\ \Rightarrow\ z_j\,\rangle \xrightarrow{t,i} s'\ \mathbf{until}\ \langle\, s_j'\ \Rightarrow\ z_j\,\rangle}\ (\text{u8})$$

Figure 3.3: Operational semantics for RT-FRP: updating rules

The evaluation of **let-signal** and **until** (rules e6 and e7) is straightforward with one exception: the value returned by s **until** $\langle s_j \Rightarrow z_j \rangle$ does not depend on any of the events $\{s_j\}$, i.e. when an event s_k occurs, s **until** $\langle s_j \Rightarrow z_j \rangle$ still returns the value of s. Such is called *delayed switching*. An alternative design is *immediate switching*, where when s_k occurs, s **until** $\langle s_j \Rightarrow z_j \rangle$ returns the value of the behavior just being switched into.

The reason we pick the delayed switching design is to allow the user to define signals that react to themselves, such as

let snapshot x $\leftarrow s$ **until** $\langle$**ext** $(\mathsf{x} > 10) \Rightarrow z \rangle$
in ext x,

which is initially s and switches to the signal z when the value of itself exceeds 10. This program will get stuck if executed using the immediate switching semantics.

Note that **ext** $(\mathsf{x} > 10)$ is a Boolean signal, but is used as an event here. We can do this because the Boolean type is just synonym of "Maybe ()" in BL (see Section 2.1), which means any Boolean signal is automatically an event of unit.

The updating rules for **let-signal** and **until** are more involved. For the construct

$$\textbf{let signal } \{z_j(x_j) = u_j\} \textbf{ in } s,$$

the signal definition group $\{z_j(x_j) = u_j\}$ is unchanged, and s is executed with the signal environment extended with the new definitions (rule u7). For the construct

$$s \textbf{ until } \langle s_j \Rightarrow z_j \rangle,$$

the events $\langle s_j \rangle$ are tested one after another until the first occurrence, then the signal evolves to the definition of the corresponding signal, with its formal parameter replaced by the value of the event (rule u7). If no event occurs, then we get back the original term with the sub-terms updated (u8).

Please be reminded that we require that all signal variables in an RT-FRP program are distinct, otherwise the rules u6 and u7 will not work. For example, if we allow the program

$$\begin{aligned}
&\textbf{let signal } \mathsf{z1}(_) = s_1 \textbf{ until } ev \Rightarrow \mathsf{z2} \\
&\qquad\qquad \mathsf{z2}(_) = s_2 \textbf{ until } ev \Rightarrow \mathsf{z1} \\
&\textbf{in let signal } \mathsf{z2}(_) = s_3 \textbf{ until } ev \Rightarrow \mathsf{z2} \\
&\quad \textbf{in } s_0 \textbf{ until } ev \Rightarrow \mathsf{z1},
\end{aligned}$$

we would expect that the inner definition of z2 will not interfere with the outer definition of the same variable, and that the z2 in the body of z1 still refers to the outer definition. Consequently, we would expect the program to exhibit the behavior

$$s_0, s_1, s_2, s_1, s_2, \cdots$$

as the event ev occurs repeatedly. However, the behavior we get by following the semantic rules will actually be

$$s_0, s_1, s_3, s_3, s_3, \cdots$$

which is not what we want. This will not happen if we rename the inner z2 to z3.

The source of this problem is that the definition of a signal variable can contain free signal variables, and thus should be represented by a closure in the environment. Indeed, we could do that and hence relax the requirement that all signal variables are distinct. However, our current representation is simpler notation-wise, and thus is used.

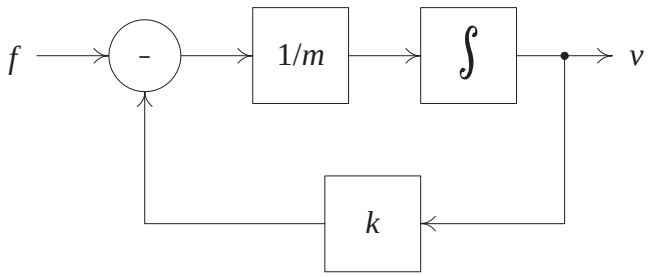
3.3 Two kinds of recursion

In a reactive system, there can be two kinds of cyclic structures that exhibit recursion in RT-FRP. One is control feedback (feeding the output of a system to its own input), and the other is cycles in a state-transition diagram. RT-FRP has different recursive constructs to

support them, which we will discuss one by one.

3.3.1 Feedback

Reactive systems often involve feedback, which corresponds to recursive **let-snapshot** in RT-FRP. For example, the velocity of a mass m under force f and friction kv can be described by the block diagram:



or more concisely by the recursive integral equation:

$$v = \int (f - kv)/m \, \mathrm{d}t$$

The RT-FRP encoding for this signal is simply:

$$\textbf{let snapshot}\ \mathsf{v} \leftarrow \textbf{integral}\ (\textbf{ext}\ (\mathsf{f} - \mathsf{k} \cdot \mathsf{v})/\mathsf{m})\textbf{in ext}\ \mathsf{v}$$

where **integral** will be defined as syntactic sugar in Section 3.7.

An important use of feedback is to add *local state* to a system. For example, a general expression of the form:

$$\textbf{let snapshot}\ x \leftarrow \textbf{delay}\ e\ s_1\ \textbf{in}\ s_2$$

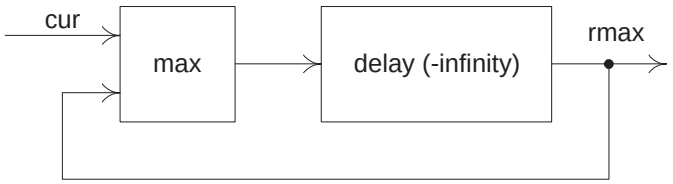
defines a signal with a local state x. The initial value of x is that of e. In each time step after that, the new state (i.e. the new value of x) is the previous value returned by s_1, which can depend on the previous value of x. As a more concrete example, we can define the running maximum of a signal s as follows:

let snapshot cur $\leftarrow s$ **in**

let snapshot rmax $\leftarrow$ **delay** $(-\infty)$ (**ext** max(cur, rmax)) **in**

ext rmax

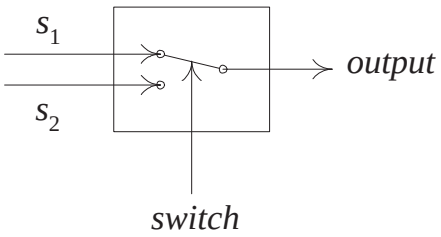
where max is a BL function that returns the larger of two real numbers. The definition is illustrated by the diagram:



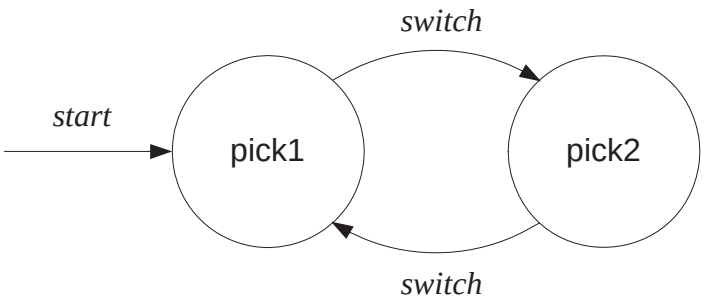
rmax is $-\infty$ in the initial step. At the $(n+1)$-th step, it is updated to be the larger one of its previous value and the value of s at step n. Therefore rmax records the maximum value of s up to the previous step.

3.3.2 Cycles in a state transition diagram

RT-FRP provides two constructs, namely **let-signal** and **until**, that allow the definition of *multi-modal* signals, that is, signals that shift from one operating mode to another with the occurrences of events. For example, here is a multiplexer that switches between two signals s_1 and s_2 whenever the event $switch$ occurs:



The state-transition diagram of this multiplexer is:



and it can be encoded in RT-FRP by the following program:

$$
\begin{array}{l}
\textbf{let snapshot}\ \mathsf{i1} \leftarrow s_1\ \textbf{in} \\
\textbf{let snapshot}\ \mathsf{i2} \leftarrow s_2\ \textbf{in} \\
\textbf{let snapshot}\ \mathsf{s} \leftarrow switch\ \textbf{in} \\
\textbf{let signal}\ \ \mathsf{pick1}(\_) = (\textbf{ext}\ \mathsf{i1})\ \textbf{until}\ \langle \textbf{ext}\ \mathsf{s} \Rightarrow \mathsf{pick2} \rangle \\
\qquad\qquad\qquad \mathsf{pick2}(\_) = (\textbf{ext}\ \mathsf{i2})\ \textbf{until}\ \langle \textbf{ext}\ \mathsf{s} \Rightarrow \mathsf{pick1} \rangle\ \textbf{in} \\
(\textbf{ext}\ \mathsf{i1})\ \textbf{until}\ \langle \textbf{ext}\ \mathsf{s} \Rightarrow \mathsf{pick2} \rangle.
\end{array}
$$

As shown in the above example, the **let-signal** definitions can be mutually recursive, in which case the corresponding state transition diagram of the system is cyclic.

3.3.3 Recursion in FRP

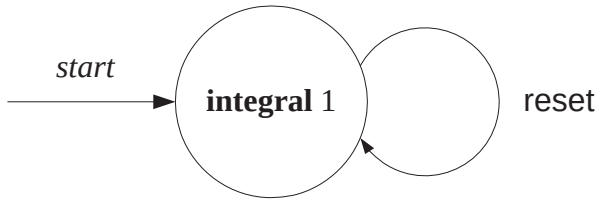
Those familiar with FRP might recall that there is no syntactic distinction between the two forms of recursion in FRP. For example, the FRP program

$$\mathsf{v} = \textbf{integral}\ ((\mathsf{f} - \mathsf{k} \cdot \mathsf{v})/\mathsf{m})$$

implements the velocity-of-mass example shown at the beginning of this section, and uses recursion for feedback; whereas the FRP program

$$\mathsf{timer} = (\textbf{integral}\ 1)\ \textbf{until}\ \mathsf{reset} \multimap \mathsf{timer}$$

(assuming suitable definition for the event reset) uses recursion for switching back to a mode, as shown below:



Yet, in both cases the same recursion syntax is used. We feel that this in practice can create confusion about the program. For example, it is not obvious what the following FRP program means:

$$u = (\textbf{integral}\ ((f - k \cdot u)/m))\ \textbf{until}\ \mathsf{reset} \Rightarrow u.$$

In RT-FRP, these two forms of recursion are clearly distinguished using different language constructs: a recursive **let-snapshot** introduces a feedback, while **let-signal** is used to define recursive switching. Therefore, the two examples are written in RT-FRP as

$$\textbf{let snapshot}\ v \leftarrow \textbf{integral}\ (\textbf{ext}\ ((f - k \cdot v)/m))$$

and

$$\textbf{let signal}\ \mathsf{timer}(\_) = \textbf{integral}\ 1\ \textbf{until}\ \langle \textbf{ext}\ \mathsf{reset} \Rightarrow \mathsf{timer} \rangle$$

respectively, where **integral** is defined as syntactic sugar in Section 3.7.2. We feel this is an improvement over the FRP syntax.

3.4 Type system

Although we have already defined the syntax and semantics of RT-FRP, the specification is not complete without a type system. This section explains why we need a type system for RT-FRP, what it is like, and what the intuition behind it is.

3.4.1 What can go wrong?

RT-FRP supports two forms of recursion [12]: *feedback* as defined by **let-snapshot**, and *recursive switching* as defined by **let-signal** and **until**. Unrestricted use of these two forms of recursion can cause programs to get stuck or use unbounded space at run-time.

Getting stuck

Untyped programs that use **let-snapshot** can get stuck. For example, evaluating

$$\textbf{let snapshot } \mathsf{x} \leftarrow \textbf{ext } \mathsf{x} \textbf{ in ext } \mathsf{x}$$

requires evaluating **ext** x in an environment where x is not bound (rule e5), and hence gets stuck. The essence of this problem is that for a signal **let snapshot** $x \leftarrow s_1$ **in** s_2, any occurrence of x in s_1 should not be needed during the evaluation of s_1, although it could be used during updating s_1. In fact, the distinction between evaluation and updating exists primarily so that evaluation can be used to "bootstrap" the recursion in an early phase, so that we can give a sensible notion of updating to such expressions.

A major design goal for the RT-FRP type system is to ensure that the above situation does not arise at run time.

Using unbounded space

In a switcher s **until** $\langle s_j \Rightarrow z_j \rangle$, if the initial signal s contains a free signal variable, then the term might grow arbitrarily large during execution. One example is the program fragment

$$\begin{aligned}&\textbf{let signal } \mathsf{z}(\_) = (s_0 \textbf{ until } ev_0 \Rightarrow \mathsf{z}) \textbf{ until } ev_1 \Rightarrow \mathsf{z}\\ &\textbf{in } s_1 \textbf{ until } ev_2 \Rightarrow \mathsf{z}\end{aligned}$$

where the initial signal for the definition of z(_) is "s_0 **until** $ev_0 \Rightarrow$ z," which has a free signal variable z.

When ev_2 occurs, this term becomes

$$\textbf{let signal } \mathsf{z}(\_) = (s_0 \textbf{ until } ev_0 \Rightarrow \mathsf{z}) \textbf{ until } ev_1 \Rightarrow \mathsf{z}$$
$$\textbf{in } (s_0 \textbf{ until } ev_0 \Rightarrow \mathsf{z}) \textbf{ until } ev_1 \Rightarrow \mathsf{z}.$$

Now if ev_0 occurs but ev_1 does not, the term in turn becomes

$$\textbf{let signal } \mathsf{z}(\_) = (s_0 \textbf{ until } ev_0 \Rightarrow \mathsf{z}) \textbf{ until } ev_1 \Rightarrow \mathsf{z}$$
$$\textbf{in } ((s_0 \textbf{ until } ev_0 \Rightarrow \mathsf{z}) \textbf{ until } ev_1 \Rightarrow \mathsf{z}) \textbf{ until } ev_1 \Rightarrow \mathsf{z}.$$

From this point on, it is clear that as long as ev_1 does not occur, every time ev_0 occurs, the program becomes larger.

For a similar reason, we do not want to allow any s_j in s **until** $\langle s_j \Rightarrow z_j \rangle$ to contain free signal variables. As an exercise, the reader can determine the condition under which the following program fragment can grow arbitrarily large:

$$\textbf{let signal } \mathsf{z}(\_) = s_0 \textbf{ until } (s_1 \textbf{ until } ev_0 \Rightarrow \mathsf{z}) \Rightarrow \mathsf{z}$$
$$\textbf{in } s_2 \textbf{ until } ev_1 \Rightarrow \mathsf{z}$$

3.4.2 Typing rules

We have seen how untyped programs can go wrong at run time. Our approach to prevent such run-time errors is to use a type system to reject those ill-behaved programs. This section presents such a type system.

Types

For simplicity of presentation, we do not explicitly distinguish between the signal types for RT-FRP and the types provided by the base language. The meaning of these types will vary depending on whether they are assigned to base language terms or RT-FRP signals. In the first case, α will have its usual interpretation. In the second case, it will mean “a

signal carrying values of type α".

Contexts

The typing of RT-FRP programs is done in two variable contexts and one signal context. The variable contexts Γ and Θ are functions from variables to types, as defined in Section 2.2.

Two variable contexts are needed to ensure that the phase distinction between the evaluation of a term and updating a term is reflected in enough detail so as to allow us to guarantee type preservation in the presence of recursion.

We will also treat context functions as sets (their graph). The reader is again advised that we require all signal variables in a program to be distinct.

A *signal context* Δ is a function from signal variables to types.

$$\text{Signal contexts} \quad \Delta \quad ::= \quad \{\, z_j : \alpha_j \to \alpha'_j \,\}.$$

A binding $z : \alpha \to \alpha'$ in Δ means that z is a signal of type α' parameterized by a variable of type α.

Typing judgment

Figure 3.4 defines the RT-FRP type system using a judgment

$$\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ s : \alpha,$$

read "s is a signal of type α in contexts Γ, Θ, and Δ." Intuitively, Γ contains variables that can be used in evaluating and updating s, while Θ contains variables that can be used in the updating, but *not* evaluation, of s. The signal context Δ assigns types to all free signal variables in s. To simplify the typing rules, we require that *all base language variables and signal variables in a program are distinct.*

The signal **input** has type Input, a placeholder for the signature of the external stimuli visible to the system. Depending on the application, Input can be instantiated to any type in the BL language.

The signal **time** has type Real. The term **ext** e is a signal of type α when e is a base language term of type α. In this typing rule, e is typed without using Δ. Intuitively, this means that signal variables cannot be exported to the base language.

The type of **delay** $e\ s$ is the type of s, given that the base language term e has the same type.

A term **let snapshot** $x \leftarrow s_1$ **in** s_2 has the same type as s_2, assuming that s_1 is well-typed. The typing reflects the fact that this is a recursive binding construct, and that x can occur anywhere.

A group of mutually recursive signals are defined by **let-signal**. A signal definition has the form $z(x) = u$, where x is the formal parameter for z, and u (a switcher) is the definition body. The type of **let signal** $\cdots$ **in** s is the same as the type of s.

In a term s **until** $\langle s_j \Rightarrow z_j \rangle$, the type of an event s_j must match the parameter type of the signal template z_j, and the result type of all z_j must be the same as the type of s. Note that none of the sub-terms s and s_j can contain free signal variables, as shown by the rule t7.

3.4.3 How it comes about

The reader might still be wondering why we designed the type system to be the way it is. This section explains some of the intuition behind our design. Uninterested readers can skip this section, as it will not affect the understanding of the rest of the thesis.

Typing of delay

The typing rule for **delay** (rule t4) has a side condition requiring that any value v typed α in context Γ can also be typed α in the empty context. This is a property of the base

$$\frac{}{\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}} \mathbf{input} : \mathsf{Input}}\ (\mathsf{t1}) \qquad \frac{}{\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}} \mathbf{time} : \mathsf{Real}}\ (\mathsf{t2})$$

$$\frac{\Gamma \vdash\ e : \alpha}{\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}} \mathbf{ext}\ e : \alpha}\ (\mathsf{t3}) \qquad \frac{\begin{array}{c}\forall v.\ \Gamma \vdash\ v : \alpha \implies \emptyset \vdash\ v : \alpha \\ \Gamma \vdash\ e : \alpha \qquad \Gamma \uplus \Theta;\ \emptyset;\ \Delta \vdash_{\mathbf{S}}\ s : \alpha\end{array}}{\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}} \mathbf{delay}\ e\ s : \alpha}\ (\mathsf{t4})$$

$$\frac{\begin{array}{c}\Gamma;\ \Theta \uplus \{x : \alpha_1\};\ \Delta \vdash_{\mathbf{S}}\ s_1 : \alpha_1 \\ \Gamma \uplus \{x : \alpha_1\};\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ s_2 : \alpha_2\end{array}}{\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}} \mathbf{let\ snapshot}\ x\ \leftarrow\ s_1\ \mathbf{in}\ s_2 : \alpha_2}\ (\mathsf{t5})$$

$$\frac{\begin{array}{c}\{\,\Gamma \uplus \{x_i : \alpha_i\};\ \Theta;\ \Delta \uplus \Delta' \vdash_{\mathbf{S}}\ u_i : \alpha'_i\,\}^{i \in J} \\ \Gamma;\ \Theta;\ \Delta \uplus \Delta' \vdash_{\mathbf{S}}\ s : \alpha \qquad \Delta' \equiv \{z_j : \alpha_j \to \alpha'_j\}^{j \in J}\end{array}}{\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}} \mathbf{let\ signal}\ \{\,z_j(x_j)\ =\ u_j\,\}^{j \in J}\ \mathbf{in}\ s : \alpha}\ (\mathsf{t6})$$

$$\frac{\Gamma;\ \Theta;\ \emptyset \vdash_{\mathbf{S}}\ s : \alpha \qquad \{\,\Gamma \uplus \Theta;\ \emptyset;\ \emptyset \vdash_{\mathbf{S}}\ s_j : \mathsf{Maybe}\ \alpha_j\,\}}{\Gamma;\ \Theta;\ \Delta \uplus \{z_j : \alpha_j \to \alpha\} \vdash_{\mathbf{S}}\ s\ \mathbf{until}\ \langle\, s_j\ \Rightarrow\ z_j\,\rangle : \alpha}\ (\mathsf{t7})$$

Figure 3.4: Type system for RT-FRP

language and the type α.

In BL, values cannot contain free variables and therefore the side condition is satisfied for all types α, as Lemma 2.1 shows. Nonetheless, this might not be the case for other base languages. For example, in a base language that supports some form of λ-calculus, the λ-abstraction "λx.x $-$ y $+$ z" is a value, but it contains free variables and therefore cannot be typed in the empty context.

The motivation for the side condition in rule t4 is to ensure that the execution of a program preserves its well-typedness. Consider the following program P where the base language supports λ-calculus:

let snapshot x $\leftarrow$ **let snapshot** f $\leftarrow$ **delay** (λy.y) (**ext** λy.x)

in ext (f 0)

in ext x

This program is well-typed if rule t4 does not have the side requirement. However, when

P is executed, it will be updated to the following program P' in the first step:

$$\begin{aligned}&\textbf{let snapshot } \mathsf{x} \leftarrow \textbf{let snapshot } \mathsf{f} \leftarrow \textbf{delay } \underline{(\lambda \mathsf{y}.\mathsf{x})}\ (\textbf{ext } \lambda \mathsf{y}.\mathsf{x})\\ &\qquad\qquad\qquad\qquad \textbf{in ext } (\mathsf{f}\ 0)\\ &\textbf{in ext } \mathsf{x}\end{aligned}$$

where the underlined part has changed. P', however, defines the value of x as $(\lambda \mathsf{y}.\mathsf{x})\ 0$, which is a cyclic definition, and therefore is not well-typed.

Typing of let-signal and until

We can say that the syntax and typing rules for **let-signal** and **until** are the essence of the RT-FRP language: they together ensure the program size to be bounded while allowing interesting recursive-switching programs. This section explains some of the intuition behind the design of these two constructs.

The typing of RT-FRP programs is done in a pair of variable environments (Γ and Θ) and a signal environment (Δ). Roughly speaking, Γ and Θ ensure that the execution of a program does not get stuck (i.e. the value of a variable is known when it is needed), while Δ ensures that the size of a program does not exceed a bound.

Therefore, the best way to understand the typing rules for **let-signal** and **until** is probably to project them into two views. First, we focus on the variable environments Γ and Θ:

$$\frac{\{\,\Gamma \uplus \{x_i : \alpha_i\};\ \Theta;\ \cdots \vdash_{\mathbf{S}}\ u_i : \alpha'_i\,\}^{i \in J} \qquad \Gamma;\ \Theta;\ \cdots \vdash_{\mathbf{S}}\ s : \alpha}{\Gamma;\ \Theta;\ \cdots \vdash_{\mathbf{S}}\ \textbf{let signal}\ \{\,z_j(x_j)\ =\ u_j\,\}^{j \in J}\ \textbf{in}\ s : \alpha}\ (\mathsf{t6})$$

$$\frac{\Gamma;\ \Theta;\ \cdots \vdash_{\mathbf{S}}\ s : \alpha \qquad \{\,\Gamma \uplus \Theta;\ \emptyset;\ \cdots \vdash_{\mathbf{S}}\ s_j : \mathsf{Maybe}\ \alpha_j\,\}}{\Gamma;\ \Theta;\ \cdots \vdash_{\mathbf{S}}\ s\ \textbf{until}\ \langle\, s_j\ \Rightarrow\ z_j\,\rangle : \alpha}\ (\mathsf{t7})$$

Now it is clear that t6 adds x_j to the current environment when typing u_j, and that t7 allows s_j to freely use variables from both the current and the delayed environment, which

corresponds to the "delayed switching" semantics for **until**.

Next we focus on the signal environment:

$$\frac{\{\cdots\,;\,\Delta \uplus \Delta' \vdash_{\mathbf{S}}\ u_i : \alpha'_i\}^{i\in J} \qquad \cdots\,;\,\Delta \uplus \Delta' \vdash_{\mathbf{S}}\ s : \alpha \qquad \Delta' \equiv \{z_j : \alpha_j \to \alpha'_j\}^{j\in J}}{\cdots\,;\Delta \vdash_{\mathbf{S}}\ \textbf{let signal}\ \{\, z_j(x_j)\ =\ u_j\,\}^{j\in J}\ \textbf{in}\ s : \alpha}\ (\mathsf{t6})$$

$$\frac{\cdots\,;\,\emptyset \vdash_{\mathbf{S}}\ s : \alpha \qquad \{\cdots\,;\,\emptyset \vdash_{\mathbf{S}}\ s_j : \mathsf{Maybe}\ \alpha_j\,\}}{\cdots\,;\,\Delta \uplus \{z_j : \alpha_j \to \alpha\} \vdash_{\mathbf{S}}\ s\ \textbf{until}\ \langle\, s_j\ \Rightarrow\ z_j\,\rangle : \alpha}\ (\mathsf{t7})$$

The rule t6 shows that signals defined in this **let-signal** group and the enclosing **let-signal** groups can both be used in s and u_j. The rule t7 says that neither s nor s_j in s **until** $\langle\, s_j\ \Rightarrow\ z_j\,\rangle$ can have any free signal variable.

We impose such restrictions because, as we have shown in Section 3.4.1, a term can grow arbitrarily large without them.

3.5 Resource bound

In this section we prove the basic resource-boundedness properties of RT-FRP, namely, that the time and space in each execution step is bounded:

Theorem 3.1 (Resource bound). For any closed and well-typed program, we know that

1. its single-step execution terminates and preserves types, and
2. there is a bound for the time and space it needs during execution.

The proof of the first part of this theorem is a special instance of Lemma 3.3. The proof of the second part requires formalizing notions of cost and showing that they are bounded during the execution of any program. In the rest of this section, we present the technical details required to establish this result.

3.5.1 Type preservation and termination

This section establishes some basic properties of RT-FRP useful for proving its resource-boundedness. Uninterested readers can skip this section.

Compatibility

In order to express the type safety property concisely, we must have a concise way for expressing the necessary constraints on the environments involved in the statement of the properties. In general, we will need to assume that an environment $\mathcal{X}$ is compatible with a context Γ. We write this relation as $\Gamma \vdash \mathcal{X}$, which was defined in Section 2.3.

Similarly, we define $\Gamma;\ \Theta;\ \Delta \vdash \mathcal{Z}$ (read "the signal environment $\mathcal{Z}$ is compatible with the contexts Γ, Θ, and Δ") as follows:

$$\boxed{\Gamma;\ \Theta;\ \Delta \vdash \mathcal{Z}} \qquad \frac{\{\Gamma \oplus \{x_i :\ \alpha_i\};\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ u_i : \alpha_i'\}^{i \in J} \qquad \Delta \equiv \{z_j : \alpha_j \to \alpha_j'\}^{j \in J}}{\Gamma;\ \Theta;\ \Delta \vdash \{z_j \mapsto \lambda x_j.u_j\}^{j \in J}}.$$

It is easy to show the above definition enjoys the following forms of weakening:

Lemma 3.1 (Basic properties).

$$\frac{}{\Gamma;\ \Theta;\ \emptyset \vdash \emptyset} \qquad \frac{\Gamma \vdash \mathcal{X} \quad \Gamma \vdash\ v : \alpha}{\Gamma \oplus \{x : \alpha\} \vdash \mathcal{X} \oplus \{x \mapsto v\}}$$

$$\frac{\Gamma;\ \Theta;\ \Delta \vdash \mathcal{Z} \quad \Gamma;\ \Theta;\ \Delta' \vdash \mathcal{Z}'}{\Gamma;\ \Theta;\ \Delta \uplus \Delta' \vdash \mathcal{Z} \uplus \mathcal{Z}'} \qquad \frac{\Gamma;\ \Theta \uplus \{x : \alpha\};\ \Delta \vdash s : \alpha'}{\Gamma \uplus \Theta;\ \{x : \alpha\};\ \Delta \vdash s : \alpha'}$$

Assumptions about the base language

In order to prove the key properties of RT-FRP, we must be explicit about our assumptions about the base language. We assume three key properties of the base language: First, that evaluation terminates and preserves typing. Second, that values of Maybe α types either

are Nothing or have the form Just v. Third, that the type system enjoys substitutivity. These requirements are formalized as follows:

$$\frac{\Gamma \vdash \mathcal{X} \quad \Gamma \vdash e : \alpha}{\exists v.\, (\, \mathcal{X} \;\vdash\; e \hookrightarrow v \quad \Gamma \vdash\; v : \alpha\,)}$$

$$\frac{\Gamma \vdash\; v : \mathsf{Maybe}\ \alpha}{v \equiv \mathsf{Nothing} \text{ or } \exists v'.\, (\, v \equiv \mathsf{Just}\ v' \quad \Gamma \vdash\; v' : \alpha\,)}$$

$$\frac{\Gamma \vdash\; e : \alpha \quad \Gamma \uplus \{x : \alpha\} \vdash\; e' : \alpha'}{\Gamma \vdash\; e'[x := e] : \alpha'}$$

It is easily verified that BL satisfies these properties.

Type preservation and termination

Using the substitutivity assumption about the base language, it is now possible to establish the following lemma for RT-FRP:

Lemma 3.2 (Substitution). Whenever $\Gamma \vdash\; v : \alpha$ and $\Gamma \uplus \{x : \alpha\};\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ s : \alpha'$, we have $\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ s[x := v] : \alpha'$.

Next, we show this important lemma:

Lemma 3.3 (Type preservation and termination). For all Γ, Θ, Δ, $\mathcal{X}$, $\mathcal{Z}$, s, and α,

1. if $\Gamma \vdash \mathcal{X}$ and $\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ s : \alpha$, then there exists a value v such that $\mathcal{X} \;\vdash\; s \overset{t,i}{\hookrightarrow} v$ and $\Gamma \vdash\; v : \alpha$, and

2. if $\Gamma \uplus \Theta \vdash \mathcal{X}$, and $\Gamma;\ \Theta;\ \Delta \vdash \mathcal{Z}$ and $\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ s : \alpha$, then there exists a term s' such that $\mathcal{X};\ \mathcal{Z} \;\vdash\; s \xrightarrow{t,i} s'$ and $\Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ s' : \alpha$.

Proof. The proof of both parts is by induction over the height of the typing derivation. The proof of the first part uses the assumptions about the base language to establish the

soundness of the **ext** construct. The proof of the second part uses the substitutivity property, and the first part of the lemma. □

After this property is established, it is easy to see that not only is evaluation always terminating, but it is also deterministic whenever the base language is also deterministic.

3.5.2 Resource boundedness

Having established that the language is terminating, we now come to the proof of the second part of our main theorem, namely, time- and space-boundedness. As a measure of the time and space needed for executing a program, we will use term size at run-time (modulo the size of base language terms). This measure is reasonable because of two observations: First, the size of a derivation tree is bounded by the term size. Second, the derivation is syntax directed, and so the time needed to propagate the information around a rule is bounded by the size of the term (assuming a naïve implementation, where each term is copied in full). Thus, our focus will be on showing that there exists a measure on programs that does not increase during run-time.

We formally define the *size of a term* s, written $|s|$, to be the number of constructors, base language expressions, and signals in the term:

$$\begin{aligned}
|\textbf{input}| &= 1 \\
|\textbf{time}| &= 1 \\
|\textbf{ext}\ e| &= 2 \\
|\textbf{delay}\ e\ s| &= 2 + |s| \\
|\textbf{let snapshot}\ x \leftarrow s_1\ \textbf{in}\ s_2| &= 2 + |s_1| + |s_2| \\
\left|\textbf{let signal}\ \{\, z_j(x_j) = u_j \,\}^{j \in \mathbb{N}_n}\ \textbf{in}\ s\right| &= 1 + 2n + \Sigma_{j=1}^{n} |u_j| + |s| \\
\left|s\ \textbf{until}\ \langle\, s_j \Rightarrow z_j \,\rangle^{j \in \mathbb{N}_n}\right| &= 1 + n + |s| + \Sigma_{j=1}^{n} |s_j|.
\end{aligned}$$

We must show the well-formedness of the definition of $|-|$:

Lemma 3.4. $|s|$ is defined for all s.

The proof is by induction on s and can be found in Appendix A.1.

As mentioned earlier, the size of a term can grow at run-time. However, we can still show that there exists a bound for the size of the term at run-time. The following function on terms (where m is an integer) will be used to define such a bound:

$$\begin{aligned}
\|\textbf{input}\|_m &= 1 \\
\|\textbf{time}\|_m &= 1 \\
\|\textbf{ext}\ e\|_m &= 2 \\
\|\textbf{delay}\ e\ s\|_m &= 2 + \|s\|_m \\
\|\textbf{let snapshot}\ x \leftarrow s_1\ \textbf{in}\ s_2\|_m &= 2 + \|s_1\|_m + \|s_2\|_m
\end{aligned}$$

$$\begin{aligned}
&\left\|\textbf{let signal}\ \{\, z_j(x_j) = u_j \,\}^{j\in\mathbb{N}_n}\ \textbf{in}\ s\right\|_m \\
= \ & 1 + 2n + \Sigma_{j=1}^{n}|u_j| + \|s\|_{\max\{\|\{z_j\mapsto\lambda x_j.u_j\}^{j\in\mathbb{N}_n}\|,\, m\}}
\end{aligned}$$

$$\begin{aligned}
&\left\|s\ \textbf{until}\ \langle\, s_j \Rightarrow z_j \,\rangle^{j\in\mathbb{N}_n}\right\|_m \\
= \ & \max\left\{1 + n + \|s\|_0 + \Sigma_{j=1}^{n}\|s_j\|_0,\, m\right\}
\end{aligned}$$

where $\|\{z_j \mapsto \lambda x_j.u_j\}\|$ is given by:

$$\|\{z_j \mapsto \lambda x_j.u_j\}\| = \max\left(\{0\} \cup \left\{\|u_j\|_0\right\}\right).$$

The idea is that assuming m is the size bound for the free signal variables in s, $\|s\|_m$ gives the size bound for s.

Since **input**, **time**, and **ext** e do not change during execution, their size bounds are simply their sizes. The terms **delay** and **let-snapshot** are updated by updating their respective sub-terms. This explains the definition of $\|\textbf{delay}\ e\ s\|_m$ and $\|\textbf{let snapshot}\ x \leftarrow s_1\ \textbf{in}\ s_2\|_m$.

When we update the term $s' \equiv \textbf{let signal}\ \{z_j(x_j) = u_j\}\ \textbf{in}\ s$, the signal variable definitions are not touched, and we just update the sub-term s. To understand the definition of $\|\textbf{let signal}\ \{z_j(x_j) = u_j\}\ \textbf{in}\ s\|_m$, we consider an arbitrary free signal variable z in s. We know that either z is a free variable in s', in which case m is a bound for z, or $z \equiv z_j$ for

some j, in which case $\max\left\{\left\|\{z_j \mapsto \lambda x_j.u_j\}^{j\in\mathbb{N}_n}\right\|, m\right\}$ is a bound for z. In neither case can z grow larger than $\max\left\{\left\|\{z_j \mapsto \lambda x_j.u_j\}^{j\in\mathbb{N}_n}\right\|, m\right\}$.

To define $\|s'\|_m$ where $s' \equiv s$ **until** $\langle\, s_j \Rightarrow z_j \,\rangle^{j\in\mathbb{N}_n}$, we note that s and s_j cannot contain free signal variables. Therefore, when no switching occurs, s' cannot grow larger than $1 + n + \|s\|_0 + \Sigma_{j=1}^{n}\|s_j\|_0$, and when a switching to some z_j occurs, s' cannot grow larger than m since m (by definition) is an upper bound for z_j. Hence, the bound for s' is the maximum of $1 + n + \|s\|_0 + \Sigma_{j=1}^{n}\|s_j\|_0$ and m.

Of course, we need to prove the well-definedness of $\|s\|_m$:

Lemma 3.5. For all term s and integer m, $\|s\|_m$ is defined.

The proof is again by induction on s and can be found in Appendix A.1.

In order to establish the bound, we must always consider a term in the context of a particular signal environment. Thus we define the *term size bound* for a term s under signal environment $\mathcal{Z}$ to be $\|s\|_{\|\mathcal{Z}\|}$. First, it is useful to know that this measure is an upper bound for term size $|s|$:

Lemma 3.6. For all term s and integer m, $|s| \leq \|s\|_m$.

This lemma can be proved by induction on s. The full proof is presented in Appendix A.1.

Now we can show that even though term size can grow during execution, its size bound as defined by $\|-\|_{\|\mathcal{Z}\|}$ does not:

Lemma 3.7 (Bound preservation). $\mathcal{X};\, \mathcal{Z} \vdash s \xrightarrow{t,i} s'$ implies $\|s'\|_{\|\mathcal{Z}\|} \leq \|s\|_{\|\mathcal{Z}\|}$.

The proof is by induction on the derivation for $\mathcal{X};\, \mathcal{Z} \vdash s \xrightarrow{t,i} s'$, and can be found in Appendix A.1.

3.5.3 Space and time complexity of terms

A natural question to ask is: for a program s of size n (i.e. $|s| = n$), how large can $\|s\|_m$ be? Is it $O(n)$, $O(n^2)$, $O(2^n)$, or something else? We find that the worst case of $\|s\|_m$ is

exponential with respect to $|s|$:

Lemma 3.8. For any term s and integer $m \geq 0$, we have that $\|s\|_m \leq \max\{1, m\} \cdot 2^{|s|}$.

The proof is by induction on s, and is presented in Appendix A.1. As the execution time is bounded by the term size, $\max\{1, m\} \cdot 2^{|s|}$ also gives the time complexity of executing the program s.

An exponentially growing program

Lemma 3.8 shows that a term can grow at most exponentially with respect to its initial size, but can this really occur? This section shows that the answer is yes by constructing such an example.

We inductively define a series of m programs $\langle P_i \rangle^{i \in \mathbb{N}_m}$, where:

$$
\begin{aligned}
P_1 \equiv\ & \textbf{let}\ \ \textbf{signal}\ z_1(\_) = s\ \textbf{until}\ \langle\rangle\ \textbf{in} \\
& \quad \textbf{let}\ \ \textbf{snapshot}\ \mathsf{x} \leftarrow (\textbf{ext}\ 0)\ \textbf{until}\ ev \Rightarrow z_1\ \textbf{in} \\
& \quad\ (\textbf{ext}\ 0)\ \textbf{until}\ ev \Rightarrow z_1
\end{aligned}
$$

and for all $k \in \mathbb{N}_{m-1}$,

$$
\begin{aligned}
P_{k+1} \equiv\ & \textbf{let}\ \ \textbf{signal}\ z_{k+1}(\_) = P_k\ \textbf{until}\ \langle\rangle\ \textbf{in} \\
& \quad \textbf{let}\ \ \textbf{snapshot}\ \mathsf{x} \leftarrow (\textbf{ext}\ 0)\ \textbf{until}\ ev \Rightarrow z_{k+1}\ \textbf{in} \\
& \quad\ (\textbf{ext}\ 0)\ \textbf{until}\ ev \Rightarrow z_{k+1}
\end{aligned}
$$

where s and ev are defined as

$s \equiv \textbf{ext}\ 0$, and

$ev \equiv \textbf{delay}\ (\mathsf{Just}\ ())\ (\textbf{ext}\ \mathsf{Nothing})$

Note that ev is an event that occurs once in the initial step, and then never occurs again.

By the definition of $|-|$, we can calculate the sizes of the programs as:

$$\begin{aligned}
|P_1| &= 3 + |s \textbf{ until } \langle\rangle| + |\textbf{let snapshot } \mathsf{x} \leftarrow (\textbf{ext } 0) \textbf{ until } ev \Rightarrow z_1 \textbf{ in } (\textbf{ext } 0) \textbf{ until } ev \Rightarrow z_1| \\
&= 3 + (1 + |s|) + (2 + |(\textbf{ext } 0) \textbf{ until } ev \Rightarrow z_1| + |(\textbf{ext } 0) \textbf{ until } ev \Rightarrow z_1|) \\
&= 6 + |s| + 2 \cdot |(\textbf{ext } 0) \textbf{ until } ev \Rightarrow z_1| \\
&= 6 + |s| + 2 \cdot (2 + |\textbf{ext } 0| + |ev|) \\
&= 6 + |s| + 2 \cdot (2 + 2 + 4) \\
&= 22 + |s|
\end{aligned}$$

We notice that P_{k+1} has the same structure as P_1, except that P_k is substituted for s. Hence we have

$$|P_{k+1}| = 22 + |P_k|$$

Considering the initial condition $|P_1| = 22 + |s|$, we get that

$$|P_k| = 22k + |s| \quad \text{for all } k \in \mathbb{N}_m.$$

Let P_k^i be the new program obtained by executing P_k for i steps. Apparently

$$\begin{aligned}
P_1^1 \equiv \textbf{let } &\textbf{ signal } z_1(_) = s \textbf{ until } \langle\rangle \textbf{ in} \\
&\textbf{let snapshot } \mathsf{x} \leftarrow s \textbf{ until } \langle\rangle \textbf{ in} \\
&s \textbf{ until } \langle\rangle
\end{aligned}$$

and for all $k \in \mathbb{N}_{m-1}$,

$$\begin{aligned}
P_{k+1}^1 \equiv \textbf{let } &\textbf{ signal } z_{k+1}(_) = P_k \textbf{ until } \langle\rangle \textbf{ in} \\
&\textbf{let snapshot } \mathsf{x} \leftarrow P_k \textbf{ until } \langle\rangle \textbf{ in} \\
&P_k \textbf{ until } \langle\rangle
\end{aligned}$$

What's more, for all $k \in \mathbb{N}_{m-1}$ and $i \in \mathbb{N}_k$, we have

$$P_{k+1}^{i+1} \equiv \textbf{let signal } z_{k+1}(\_) = P_k \textbf{ until } \langle\rangle \textbf{ in}$$
$$\textbf{let snapshot } \mathsf{x} \leftarrow P_k^i \textbf{ until } \langle\rangle \textbf{ in}$$
$$P_k^i \textbf{ until } \langle\rangle$$

Now we can inspect the sizes of the new programs:

$$\begin{aligned}
\left|P_1^1\right| &= 3 + |s \textbf{ until } \langle\rangle| + |\textbf{let snapshot } \mathsf{x} \leftarrow s \textbf{ until } \langle\rangle \textbf{ in } s \textbf{ until } \langle\rangle| \\
&= 3 + (1 + |s|) + (2 + |s \textbf{ until } \langle\rangle| + |s \textbf{ until } \langle\rangle|) \\
&= 3 + (1 + |s|) + (2 + 2 \cdot (1 + |s|)) \\
&= 8 + 3|s| \\
\left|P_{k+1}^1\right| &= 8 + 3|P_k| \\
&= 66k + 8 + 3|s|
\end{aligned}$$

Therefore for all $k \in \mathbb{N}_m$,

$$P_k^1 = 66k - 58 + 3|s|$$

We also have that for all $k \in \mathbb{N}_{m-1}$ and $i \in \mathbb{N}_k$,

$$\begin{aligned}
\left|P_{k+1}^{i+1}\right| &= 3 + |P_k \textbf{ until } \langle\rangle| + \left|\textbf{let snapshot } \mathsf{x} \leftarrow P_k^i \textbf{ until } \langle\rangle \textbf{ in } P_k^i \textbf{ until } \langle\rangle\right| \\
&= 3 + (1 + |P_k|) + (2 + \left|P_k^i \textbf{ until } \langle\rangle\right| + \left|P_k^i \textbf{ until } \langle\rangle\right|) \\
&= 3 + (1 + 22k + |s|) + (2 + 2 \cdot (1 + \left|P_k^i\right|)) \\
&= 22k + 8 + |s| + 2\left|P_k^i\right|
\end{aligned}$$

Therefore,

$$\begin{aligned}
|P_m^m| &= 22m - 14 + |s| + 2\left|P_{m-1}^{m-1}\right| \\
&= 22m - 14 + |s| + 2 \cdot (22(m-1) - 14 + |s| + 2\left|P_{m-2}^{m-2}\right|) \\
&\vdots \\
&= 2^{m-1}\left|P_1^1\right| + \Sigma_{i=0}^{m-2} 2^i (22(m-i) - 14 + |s|) \\
&= 2^{m-1}\left|P_1^1\right| + (22m - 14 + |s|) \cdot \Sigma_{i=0}^{m-2} 2^i - \Sigma_{i=0}^{m-2} i \cdot 2^i \\
&= 2^{m-1}(8 + 3|s|) + (22m - 14 + |s|) \cdot (2^{m-1} - 1) - \Sigma_{i=0}^{m-2} i \cdot 2^i
\end{aligned}$$

Let $a = \Sigma_{i=0}^{m-2} i \cdot 2^i$, we have

$$\begin{aligned}
a &= 2a - a \\
&= \Sigma_{i=0}^{m-2} i \cdot 2^{i+1} - \Sigma_{i=0}^{m-2} i \cdot 2^i \\
&= \Sigma_{i=1}^{m-1} (i-1) \cdot 2^i - \Sigma_{i=0}^{m-2} i \cdot 2^i \\
&= \Sigma_{i=1}^{m-1} (i-1) \cdot 2^i - \Sigma_{i=1}^{m-2} i \cdot 2^i \\
&= (m-2) \cdot 2^{m-1} + \Sigma_{i=1}^{m-2} (i - 1 - i) \cdot 2^i \\
&= (m-2) \cdot 2^{m-1} - \Sigma_{i=1}^{m-2} 2^i \\
&= (m-2) \cdot 2^{m-1} - (2^{m-1} - 2) \\
&= (m-3) \cdot 2^{m-1} + 2
\end{aligned}$$

Hence,

$$\begin{aligned}
|P_m^m| &= 2^{m-1}(8 + 3|s|) + (22m - 14 + |s|) \cdot (2^{m-1} - 1) - \Sigma_{i=0}^{m-2} i \cdot 2^i \\
&= 2^{m-1}(8 + 3|s|) + (22m - 14 + |s|) \cdot (2^{m-1} - 1) - ((m-3) \cdot 2^{m-1} + 2) \\
&= 2^{m-1}((8 + 3|s|) + (22m - 14 + |s|) - (m-3)) - ((22m - 14 + |s|) + 2) \\
&= 2^{m-1}(21m - 3 + 4|s|) - 22m + 12 - |s| \\
&= 2^{m-1}(21m + 5) - 22m + 10
\end{aligned}$$

In other words, after m steps of execution, the program P_m grows to size $2^{m-1}(21m+5) - 22m + 10$, which is exponential with respect to m. Since $|P_m|$ is linear with respect to m,

we know that $|P_m^m|$ is indeed exponential with respect to $|P_m|$.

However, although the worst case is exponential, we only expect to see it in contrived examples. For a normal program, our experience is that the upper bound is linear or close to linear to the size of the program.

3.6 Expressiveness

To prevent unlimited memory usage, synchronous data-flow reactive languages [52, 18, 7, 5] adopt the model that all signals are statically allocated: the number of active signals remains fixed through the lifespan of the system. This over-simplification makes it difficult to express systems that change modes dynamically, where the system performs different tasks in different modes.

To address this problem, the language Synchronous Kahn Networks [8] was proposed. This language basically extends LUSTRE with recursive switching and higher-order functions. When a signal switches from one mode to another, new local signals can be introduced, hence breaking the static allocation limit. However, unwise use of this feature may cause the number of signals to keep growing, rendering this language unsuitable for real-time systems. FRP allows general recursion, and therefore does not match the need for developing real-time reactive systems either.

We have shown that RT-FRP programs are bounded in space and time consumption, yet RT-FRP still allows certain (restricted) forms of recursion. This is an improvement over existing languages. In particular, compared with synchronous data-flow reactive languages, expressing finite automata is much easier and more natural in RT-FRP. We will present a scheme for encoding arbitrary finite automata, and use a cruise control system as an example.

3.6.1 Encoding finite automata

Let M be a finite automaton with n states $z_1, \cdots, z_n$, where z_1 is the start state. When the system is in state z_j (where $j \in \mathbb{N}_n$), it behaves like s_j. From state z_j, there are m_j out-going transitions: when event $ev_{j,i}$ occurs (where $i \in \mathbb{N}_{m_j}$), M jumps from state z_j to $z_{j,i}$. Such is a generic description of a finite automaton. This automaton can be expressed in RT-FRP as

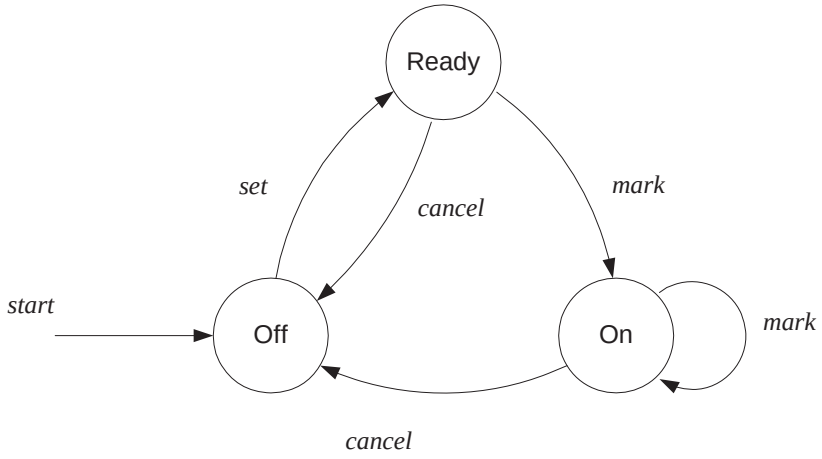
$$\begin{aligned}
&\textbf{let signal } z_1(_) = s_1 \textbf{ until } \langle ev_{1,i} \Rightarrow z_{1,i} \rangle^{i \in \mathbb{N}_{m_1}} \\
&\qquad\qquad \vdots \\
&\qquad z_n(_) = s_n \textbf{ until } \langle ev_{n,i} \Rightarrow z_{n,i} \rangle^{i \in \mathbb{N}_{m_n}} \\
&\textbf{in } s_1 \textbf{ until } \langle ev_{1,i} \Rightarrow z_{1,i} \rangle^{i \in \mathbb{N}_{m_1}} .
\end{aligned}$$

Actually, RT-FRP is capable of encoding generalized finite automata where each state is parameterized by a variable whose value is determined by a run-time event:

$$\begin{aligned}
&\textbf{let signal } z_1(x) = s_1 \textbf{ until } \langle ev_{1,i} \Rightarrow z_{1,i} \rangle^{i \in \mathbb{N}_{m_1}} \\
&\qquad\qquad \vdots \\
&\qquad z_n(x) = s_n \textbf{ until } \langle ev_{n,i} \Rightarrow z_{n,i} \rangle^{i \in \mathbb{N}_{m_n}} \\
&\textbf{in } s_1[x := v] \textbf{ until } \langle ev_{1,i}[x := v] \Rightarrow z_{1,i} \rangle^{i \in \mathbb{N}_{m_1}} .
\end{aligned}$$

3.6.2 Example: a cruise control system

As an example of encoding a finite automaton, we develop a simplified cruise control system for an automobile. The user can manipulate the system using three events: set, $mark$, and $cancel$. The system can be in one of the three states: Off, Ready, and On. The output of this system is either Nothing (if the cruise control is not active *or* the user hasn't set the desired speed yet) or Just r (if the cruise control is active *and* the user has set the desired speed to r). The state-transition diagram for this system is shown below:



Initially the system is in the Off mode. It enters the Ready mode when the user presses the set button. However, in this mode the output is still Nothing. The user can then fire the $mark$ event, which carries a real number, to notify the system of the desired speed. Once a $mark$ event is fired, the system enters the On mode, and the desired speed is set to the occurrence value of $mark$. The user can change the desired speed whenever he feels like it by firing consequent $mark$ events. The user can turn off the cruise control (i.e. return the system to the Off mode) at any time by pressing the $cancel$ button.

As we have shown, modes in controllers can easily be mapped to a **let-signal** construct in RT-FRP. Following the scheme we presented in the previous section, we can encode this controller (with suitable definitions for set, $mark$, and $cancel$) as

$$
\begin{aligned}
\textbf{let signal}\ \ \mathsf{Off}(\_) &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle set \Rightarrow \mathsf{Ready}\rangle \\
\mathsf{Ready}(\_) &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\mathsf{On}(\mathsf{x}) &= (\textbf{ext}\ (\mathsf{Just\ x}))\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\textbf{in}\ (\textbf{ext}\ \mathsf{Nothing})\ &\textbf{until}\ \langle set \Rightarrow \mathsf{Ready}\rangle.
\end{aligned}
$$

3.7 Syntactic sugar

It may be surprising to a reader familiar with FRP that some of its basic constructs are missing in RT-FRP. Many of these constructs, however, are definable in RT-FRP. By keeping the RT-FRP language simple and defining features as syntactic sugar, we have made the study of its resource-boundedness property tractable.

This section extends the RT-FRP language with some convenient syntactic sugar. In what follows, we use the notation $[\![-]\!]$ for the de-sugaring function that turns a sugared program into a plain RT-FRP program.

3.7.1 Lifting operators

We can define the lifting operators such as:

$$
\begin{aligned}
[\![\textbf{lift0}\ e]\!] &\equiv \textbf{ext}\ e \\
[\![\textbf{lift1}\ (\lambda x.e)\ s]\!] &\equiv \textbf{let snapshot}\ x \leftarrow s\ \textbf{in ext}\ e \\
[\![\textbf{lift2}\ (\lambda(x_1, x_2).e)\ s_1\ s_2]\!] &\equiv \textbf{let snapshot}\ x_1 \leftarrow s_1\ \textbf{in} \\
&\qquad\qquad \textbf{let snapshot}\ x_2 \leftarrow s_2\ \textbf{in ext}\ e
\end{aligned}
$$

and so on. We also allow the user to simplify

$$\textbf{lift}n\ (\lambda(x_1, \cdots, x_n).f(x_1, \cdots, x_n))\ s_1\ \cdots\ s_n$$

further to

$$\textbf{lift}n\ f\ s_1\ \cdots\ s_n.$$

Those familiar with λ-calculus will recognize that this is just η-conversion.

To illustrate, suppose we wish to define a signal representing the current time in min-

utes. We can do this by:

$$\textbf{let snapshot}\ \mathsf{t} \leftarrow \textbf{time in ext}\ (\mathsf{t}/60)$$

or more concisely with the **lift1** syntactic sugar:

$$\textbf{lift1}\ (\lambda \mathsf{x}.\, \mathsf{x}/60)\ \textbf{time}.$$

To compute externally with more than one signal, we have to snapshot each one separately. For example, the term:

$$\begin{array}{l}\textbf{let snapshot}\ \mathsf{x} \leftarrow s_1\ \textbf{in} \\ \quad \textbf{let snapshot}\ \mathsf{y} \leftarrow s_2\ \textbf{in ext}\ (\mathsf{x} + \mathsf{y})\end{array}$$

is a signal that is the point-wise sum of the signals s_1 and s_2. We can rewrite it using **lift2** as

$$\textbf{lift2}\ (\lambda(\mathsf{x}, \mathsf{y}).\mathsf{x} + \mathsf{y})\ s_1\ s_2$$

or more concisely

$$\textbf{lift2}\ (+)\ s_1\ s_2.$$

3.7.2 Some stateful constructs

As another example, the **when** operator in FRP turns a Boolean signal s into an event that occurs whenever s transitions from false to true. This useful operator can be defined using **delay**:

$$\begin{array}{rcl}[\![\textbf{when}\ s]\!] & \equiv & \textbf{let snapshot}\ x_1 \leftarrow s\ \textbf{in} \\ & & \quad \textbf{let snapshot}\ x_2 \leftarrow \textbf{delay}\ \mathsf{False}\ (\textbf{ext}\ x_1)\ \textbf{in} \\ & & \quad\quad \textbf{ext}\ (\neg x_2 \wedge x_1)\end{array}$$

where x_1 and x_2 are fresh variables.

A particularly useful stateful operation that we can express in RT-FRP is *integration* over time, defined below using the forward-Euler method:

$$
\begin{array}{rcl}
[\![\textbf{integral}\ s]\!] & \equiv & \textbf{let snapshot}\ v \leftarrow s\ \textbf{in} \\
& & \quad \textbf{let snapshot}\ \mathsf{t} \leftarrow \textbf{time in} \\
& & \quad\quad \textbf{let snapshot}\ \mathsf{st} \leftarrow \textbf{delay}\ (0,0,0) \\
& & \quad\quad\quad (\textbf{ext}\ (\textbf{case}\ \mathsf{st}\ \textbf{of}\ (\mathsf{i0},\mathsf{v0},\mathsf{t0}) \Rightarrow (\mathsf{i0} + \mathsf{v0} \cdot (\mathsf{t} - \mathsf{t0}), \mathsf{v}, \mathsf{t}))) \\
& & \quad\quad \textbf{in ext}\ (\textbf{case}\ \mathsf{st}\ \textbf{of}\ (\mathsf{i}, \_, \_) \Rightarrow \mathsf{i}).
\end{array}
$$

where v is a fresh variable.

Note that the internal state of **integral** s is a tuple (i, v, t), where i is the running integral, v is the previous value of s, and t is the previous sample time.

In addition, FRP's **never** and **once** operators generate event values that never occur and occur exactly once, respectively. They can be expressed in RT-FRP as follows:

$$
\begin{array}{rcl}
[\![\textbf{never}]\!] & \equiv & \textbf{ext}\ \mathsf{Nothing} \\
[\![\textbf{once}\ s]\!] & \equiv & \textbf{let snapshot}\ x \leftarrow s\ \textbf{in} \\
& & \quad \textbf{let snapshot}\ \mathsf{occ} \leftarrow \textbf{delay}\ \mathsf{False} \\
& & \quad\quad (\textbf{ext}\ (\textbf{case}\ x\ \textbf{of}\ \mathsf{Just}\ \_ \Rightarrow \mathsf{True}\ \textbf{else}\ \mathsf{occ}))\ \textbf{in} \\
& & \quad\quad\quad \textbf{ext}\ (\textbf{if}\ \mathsf{occ}\ \textbf{then}\ \mathsf{Nothing}\ \textbf{else}\ x)
\end{array}
$$

where x is fresh.

FRP's **till** operator, similar to the **until** construct in RT-FRP, can be translated as follows:

$$
\begin{array}{rcl}
[\![s_1\ \textbf{till}\ ev \Rightarrow \lambda x.s_2]\!] & \equiv & \textbf{let signal}\ z(x) = s_2\ \textbf{until}\ \langle\rangle \\
& & \textbf{in}\ s_1\ \textbf{until}\ ev \Rightarrow z
\end{array}
$$

where z is fresh.

3.7.3 Pattern matching

It is usual for functional languages to use pattern matching in definitions. For example, it is more convenient to write

$$\begin{aligned}&\textbf{let snapshot } (\mathsf{x}, \mathsf{y}) \leftarrow s\\ &\textbf{in ext } (\mathsf{x} + \mathsf{y})\end{aligned}$$

than

$$\begin{aligned}&\textbf{let snapshot } \mathsf{z} \leftarrow s\\ &\textbf{in ext } (\textbf{case } \mathsf{z} \textbf{ of } (\mathsf{x}, \mathsf{y}) \Rightarrow \mathsf{x} + \mathsf{y}).\end{aligned}$$

Standard techniques for translating definitions using pattern matching to pattern-matching-free definitions can be applied to RT-FRP programs.

3.7.4 Mutually recursive signals

The **let-snapshot** construct in RT-FRP defines a *self-recursive* signal. But what about *mutually recursive* signals?

It turns out that mutually recursive signals can be expressed as syntactic sugar in RT-FRP, as long as the base language has tuples. For example, to define two signals s_1 and s_2 that depend on each other, we may want to use the syntax:

$$\begin{aligned}&\textbf{let snapshot } x_1 \leftarrow s_1\\ &\qquad\qquad\quad\ x_2 \leftarrow s_2\\ &\textbf{in } s\end{aligned}$$

where x_1 and x_2 can both occur free in s_1, s_2, and s.

The translation of the above definition into RT-FRP can be:

$$
\begin{array}{l}
\textbf{let snapshot } (x_1, x_2) \leftarrow \textbf{let snapshot } y_1 \leftarrow s_1 \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{let snapshot } y_2 \leftarrow s_2 \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{ext } (y_1, y_2) \\
\textbf{in } s
\end{array}
$$

where y_1 and y_2 are two fresh variables.

In general, a group of n mutually recursive definitions:

$$
\begin{array}{l}
\textbf{let snapshot } x_1 \leftarrow s_1 \\
\qquad\qquad\qquad\quad x_2 \leftarrow s_2 \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\quad x_n \leftarrow s_n \\
\textbf{in } s
\end{array}
$$

can be rewritten as:

$$
\begin{array}{l}
\textbf{let snapshot } (x_1, x_2, \ldots, x_n) \leftarrow \textbf{let snapshot } y_1 \leftarrow s_1 \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let snapshot } y_2 \leftarrow s_2 \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let snapshot } y_n \leftarrow s_n \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{ext } (y_1, y_2, \ldots, y_n) \\
\textbf{in } s
\end{array}
$$

where y_1, y_2, …, y_n are n fresh variables.

As an example, we define a pair of signals that at each instant both get the value of the other signal from the previous instant:

$$
\begin{array}{l}
\textbf{let snapshot}\ \mathsf{x} \leftarrow \textbf{delay}\ 0\ (\textbf{ext}\ \mathsf{y}) \\
\qquad\qquad\qquad\qquad\ \mathsf{y} \leftarrow \textbf{delay}\ 1\ (\textbf{ext}\ \mathsf{x}) \\
\textbf{in}\ (\textbf{ext}\ \mathsf{x})
\end{array}
$$

which is then de-sugared to:

$$
\begin{array}{l}
\textbf{let snapshot}\ (\mathsf{x}, \mathsf{y}) \leftarrow \textbf{let snapshot}\ \mathsf{x1} \leftarrow \textbf{delay}\ 0\ (\textbf{ext}\ \mathsf{y})\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{let snapshot}\ \mathsf{y1} \leftarrow \textbf{delay}\ 1\ (\textbf{ext}\ \mathsf{x})\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{ext}\ (\mathsf{x1}, \mathsf{y1}) \\
\textbf{in}\ (\textbf{ext}\ \mathsf{x}).
\end{array}
$$

3.7.5 Repeated switching

FRP has a **switch** construct that repeatedly switches into a signal when certain events occur. It is possible to define **switch** in terms of **let-signal** and **until** as:

$$
\begin{array}{rcl}
[\![s\ \textbf{switch on}\ \langle x_j \leftarrow ev_j\ \textbf{in}\ s_j \rangle]\!] & = & \textbf{let snapshot}\ \langle y_j \leftarrow ev_j \rangle\ \textbf{in} \\
& & \qquad \textbf{let signal}\ \langle z_j(x_j) = s_j\ \textbf{until}\ \langle \textbf{ext}\ y_j \Rightarrow z_j \rangle \rangle \\
& & \qquad \textbf{in}\ s\ \textbf{until}\ \langle \textbf{ext}\ y_j \Rightarrow z_j \rangle
\end{array}
$$

where y_j and z_j are fresh.

3.7.6 Optional bindings

The **let-signal** construct allows each signal to be parameterized by a base language variable. When the parameter is not used, we allow the user to omit it.

Recall the cruise control system in Section 3.6.2:

$$
\begin{aligned}
\textbf{let signal}\ \ \mathsf{Off}(\_) &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle set \Rightarrow \mathsf{Ready}\rangle \\
\mathsf{Ready}(\_) &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad\qquad\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\mathsf{On}(\mathsf{x}) &= (\textbf{ext}\ (\mathsf{Just\ x}))\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad\qquad\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\textbf{in}\ (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ &\langle set \Rightarrow \mathsf{Ready}\rangle.
\end{aligned}
$$

In the definitions of Off and Ready, we do not actually use the formal parameter. Therefore, we allow the user to instead write

$$
\begin{aligned}
\textbf{let signal}\ \ \mathsf{Off} &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle set \Rightarrow \mathsf{Ready}\rangle \\
\mathsf{Ready} &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad\qquad\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\mathsf{On}(\mathsf{x}) &= (\textbf{ext}\ (\mathsf{Just\ x}))\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad\qquad\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\textbf{in}\ (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ &\langle set \Rightarrow \mathsf{Ready}\rangle.
\end{aligned}
$$

In the expression

$$s_1\ \textbf{switch on}\ x \leftarrow s\ \textbf{in}\ s_2,$$

x is in scope in s_2, and is bound to the occurrence value of the event s. However, often s_2 is not interested in this value, and there is no need to give it a name. If this is the case, we can write the expression as

$$s_1\ \textbf{switch on}\ s\ \textbf{to}\ s_2.$$

3.7.7 Direct use of signal variables

Looking at the cruise control example again, we notice that the body of the definition (**ext** Nothing) **until** $\langle set \Rightarrow \mathsf{Ready}\rangle$ simply repeats the definition for the Off signal variable. While the intention of the programmer is such that "initially the system is in the

Off mode", RT-FRP does not allow the user to directly write so. The purpose of such restriction is to simplify the semantics.

We can let the user use signal variables directly in his program, and then inline the definition before execution. For example, using the Off signal variable directly, one can rewrite the cruise control system in a simpler form:

$$
\begin{aligned}
\textbf{let signal}\ \ \mathsf{Off} &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle set \Rightarrow \mathsf{Ready}\rangle \\
\mathsf{Ready} &= (\textbf{ext}\ \mathsf{Nothing})\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\mathsf{On}(\mathsf{x}) &= (\textbf{ext}\ (\mathsf{Just}\ \mathsf{x}))\ \textbf{until}\ \langle mark \Rightarrow \mathsf{On}, \\
&\qquad\qquad\qquad\qquad cancel \Rightarrow \mathsf{Off}\rangle \\
\textbf{in}\ \mathsf{Off}.&
\end{aligned}
$$

As another example, the sample-and-hold register with initial value 0, which we defined in Section 3.1, can be written as

$$
\begin{aligned}
&\textbf{let signal}\ \mathsf{hold}(\mathsf{x}) = (\textbf{ext}\ \mathsf{x})\ \textbf{until}\ \langle sample \Rightarrow \mathsf{hold}\rangle \\
&\textbf{in}\ \mathsf{hold}(0).
\end{aligned}
$$

3.7.8 Local definitions in the base language

In RT-FRP, even if the base language supports local definitions, such definitions cannot be shared across expression boundaries. If we want to use a base language definition in more than one base language expression, we have to look for new ways. For example, in

$$
\begin{aligned}
&\textbf{let snapshot}\ \mathsf{z} \leftarrow \\
&\quad \textbf{ext}\ (\mathsf{sin}(\mathsf{x} + 5.0))\ \textbf{until}\ \langle \textbf{ext}\ (\mathsf{sin}(\mathsf{x} + 5.0) > 0.5) \Rightarrow s_1\rangle \\
&\textbf{in}\ (\textbf{ext}\ \mathsf{z})
\end{aligned}
$$

the computation of z contains two copies of the $\mathsf{sin}(x + 5.0)$, which is not optimized. If we define the syntactic sugar

$$[\![\textbf{let } x = e \textbf{ in } s]\!] \quad = \quad \textbf{let snapshot } x \leftarrow \textbf{ext } e \textbf{ in } s$$

where $x \notin FV(e)$, then we can write the above program fragment as

$$\begin{array}{l}\textbf{let } \mathsf{y} = \mathsf{sin}(\mathsf{x} + 5.0) \textbf{ in} \\ \quad \textbf{let snapshot } \mathsf{z} \leftarrow (\textbf{ext } \mathsf{y}) \textbf{ until } \langle \textbf{ext } (\mathsf{y} > 0.5) \Rightarrow s_1 \rangle \\ \quad \textbf{in } (\textbf{ext } \mathsf{z}).\end{array}$$

Chapter summary

RT-FRP improves over synchronous data-flow reactive languages by allowing restricted recursive switching, which makes it easy to directly express arbitrary finite automata.

A type system ensures that RT-FRP programs are resource-bounded even with recursion. The space and time complexity of executing a program is exponential with respect to its size in the worst case. For normal programs, our experience is that the complexity is linear or close to linear. With some syntactic sugar, much of the functionality in FRP can be regained in RT-FRP.

Chapter 4

Hybrid FRP

Many reactive systems are *hybrid* in the sense that they contain both discrete and continuous components. *Discrete components* are those that change states and interact with other components in steps, for example a switch or a clock; in contrast, *continuous components*, for example a speedometer or a numeric integrator, can change states or interact with other components continuously (i.e. at any point in time). Reactive programs embedded in continuously changing environments are examples of hybrid systems.

RT-FRP can be used to program such hybrid systems by representing continuous components by their samples. For example, although we are unable to know the reading of the speedometer at *all times*, we can sample it at every second. The samples then can be treated by RT-FRP as a signal. Of course, not all information from the continuous reading is present here, but if once per second is not good enough, we expect the program to generate better results by increasing the sampling rate to five times per second, and so on.

Still, this approach of relying on explicit sampling is not satisfactory. The study of high-level programming languages is about abstraction: we are not content with just finding a way to get the work done; rather, we would like to find the way that concerns us with the least amount of uninteresting details and artifacts. Abstraction helps the programmer to concentrate on the essence of the problem, and makes it less likely to make low-level mistakes. Anyone who has programmed in assembly languages knows how

distracting it can be to have to consider all the low-level details at all times.

A continuous component can be most naturally abstracted as a function from time, not a sequence of time-stamped values. And we want the programmer to be able to forget that he is sampling the continuous component. Unfortunately, RT-FRP has a discrete step semantics, and therefore fails to describe continuous components naturally. To use such a component, an RT-FRP programmer has to "discretize" it explicitly using sampling techniques. There are two problems in particular with this approach:

1. Sampling is an artifact of the limit of the programming language, and distracts the programmer from solving the problem.

2. For a hybrid system implemented using sampling, we would expect it in general to yield more accurate results as we increase the sampling rate. Formally, in the idealized situation where the sampling intervals approach zero, the output of the system should converge to some stable behavior. However, RT-FRP does not guarantee this and makes bugs in this nature hard to spot. For example, the following valid RT-FRP program $Heartbeat$:

   $$\textbf{let snapshot } \mathsf{x} \leftarrow \textbf{delay } 0 \textbf{ ext } (\mathsf{x} + 1) \textbf{ in } (\textbf{ext } \mathsf{x})$$

   counts the "heartbeats" of the system, and does not converge to anything as we increase the sampling rate.

To alleviate these problems, we provide a level of abstraction that hides the details of RT-FRP's sampling. This abstraction is captured in a language H-FRP (for *Hybrid FRP*) whose constructs can be easily defined in terms of RT-FRP. H-FRP is designed such that certain combinations of RT-FRP constructs (for example the above program) are not allowed, thus making it less likely to write programs that diverge as the sampling rate increases.

Although H-FRP is not as expressive as RT-FRP, it may still be preferred when pro-

gramming hybrid reactive systems. The reason is two-fold: first, although it is true that we cannot express certain RT-FRP programs in H-FRP, most of such programs are "junk" in the setting of hybrid systems, $Heartbeat$ being one example. Thus, being unable to write them is actually a *good* thing. Second, although sometimes we miss the expressiveness of RT-FRP, by programming in H-FRP, the programmer does not need to be concerned with sampling, and can pretend he is dealing with continuous signals. This makes the gap between the model and presentation narrower and helps to reduce the number of bugs introduced while translating the specification to the program.

4.1 Introduction and syntax

In RT-FRP, a signal of type "Maybe α" is used as an event that generates values of type α. There is no clear distinction between a behavior and an event: the latter is just a special case of the former. A syntactic distinction between behaviors and events is therefore unnecessary.

In H-FRP, a behavior and an event are decidedly different and cannot be confused: the former represents a continuous-time signal and can be viewed as a function on time, while the latter is a possible event occurrence and can be thought of as an optional time-stamped value.[1] Therefore, in H-FRP we distinguish events and behaviors in the syntax.

As in RT-FRP, H-FRP has a switching construct where a behavior can become active (be switched into) when an event occurs. Therefore, in an H-FRP program, while some behaviors are started at time 0 (when the system starts), some might start later. In general, the current value of a behavior depends not only on the current time, but also on when it was started, as in the case of integration. Therefore, a more precise intuition of a behavior should be a function from the start time *and* the current time to a value. Similarly, we should view an event as a possible occurrence between the start time and the current

[1]Those familiar with FRP might remember that FRP events are time-ordered *sequences* of occurrences, while here we view H-FRP events as *single* occurrences. We make this simplification to improve the presentation of our ideas. The reader should be assured that there is no technical difficulty to extend H-FRP events to multi-occurrences.

time.

The abstract syntax of H-FRP is defined in Figure 4.1. As in RT-FRP, e is still the base language expression.

$$
\begin{array}{rrcl}
\text{Behaviors} & B \ni b & ::= & \textbf{input} \mid \textbf{time} \mid \textbf{lift}\ (\lambda(x_1, \cdots, x_n).e)\ b_1\ \cdots\ b_n \mid \\
 & & & \textbf{integral}\ b \mid \textbf{let behavior}\ \{z_j(x_j) = w_j\}\ \textbf{in}\ b \mid w \\
\text{Switchers} & w & ::= & b\ \textbf{until}\ \langle ev_j \Rightarrow z_j \rangle \\
\text{Events} & EV \ni ev & ::= & \textbf{when}\ b \mid \textbf{liftev}\ (\lambda(x_0, \cdots, x_n).e)\ ev\ b_1\ \cdots\ b_n
\end{array}
$$

Figure 4.1: Syntax of H-FRP

As in RT-FRP, **input** and **time** are still the two primitive behaviors, the former being the system input signal and the latter being the time. The **lift** operator applies a function in the form of $\lambda(x_1, \cdots, x_n).e$ pointwise to n behaviors, and the result is one behavior. **integral** b is a Real-typed behaviors that is the integral of behavior b. **let-behavior** is analogous to RT-FRP's **let-signal** and defines a group of behaviors that can switch into each other. A switcher w is a behavior that switches to other behaviors when some events occur.

An event can have two forms: a predicate event **when** b that occurs when the Boolean behavior b changes from False to True, or a **liftev** construct that applies a function to the value of an event e and the current values of some behaviors when e occurs. **liftev** is a generalization to both of FRP's event mapping (the $\Rightarrow$ operator) and event snapshot (the **snapshot** combinator).

Since the only way to use an event ev in a behavior is to reference either ev or some event built from ev using **liftev** to trigger a switch, only the first occurrence of ev can have an impact on the program execution. This justifies our modeling an event as a potential *single* occurrence. Had H-FRP supported some constructs that make use of multiple occurrences of an event, for example **switch**, we would have to change the denotation of an

event to a time-ordered sequence of values, but this change is easily accommodated.

In the **lift** and **liftev** constructs, n can be any integer greater than or equal to 0. We adopt the convention of writing "$\lambda x.e$" for "$\lambda(x).e$" when $n = 1$, and writing "e" for "$\lambda().e$" when $n = 0$. Hence

$$\textbf{lift}\ \mathsf{True}, \quad \textbf{lift}\ \lambda(\mathsf{x}.\mathsf{x}+1)\ \textbf{time}, \quad \text{and}\ \textbf{lift}\ (\lambda(\mathsf{x},\mathsf{y}).\mathsf{x}+\mathsf{y})\ b_1\ b_2$$

are all syntactically valid behavior expressions.

The **integral** construct is particularly useful, as many hybrid systems are defined using integral equations. In particular, hybrid automata [22] are defined as mode switching machines where inside each mode, the behavior of the system is described by its rate of change. Hence the system state within a mode is obtained by integrating over its change rate. Being able to express integration over time directly allows us to implement such systems more naturally.

As with RT-FRP, we require that all behavior variables have distinct names. A program that reuses names can easily be made to conform with our requirement by renaming those offending variables.

4.2 Type system

The type system of H-FRP is basically the same as that of RT-FRP, except that events and behaviors now belong to different type families. The type system is actually simpler than that of RT-FRP, since H-FRP does not have the recursive **let-snapshot** construct.

4.2.1 Contexts

The typing of a behavior or event is done in two contexts:

- the *variable context* Γ which maps base language variables to types, and

- the *behavior context* Ω which assigns argument types and result types to behavior variables.

The definition of Γ is the same as in Section 2.2, and Ω is defined as:

$$\Omega \quad ::= \quad \{z_j \,:\, \alpha_j \to \alpha'_j\}$$

4.2.2 Typing judgments

The type system of H-FRP can be defined using two typing judgments:

- $\Gamma;\, \Omega \vdash\ b : \mathsf{B}\ \alpha$: "$b$ is a behavior of type α in context Γ and Ω;" and
- $\Gamma;\, \Omega \vdash\ ev : \mathsf{E}\ \alpha$: "$ev$ is an event of type α in context Γ and Ω."

The reader can find the complete typing rules in Figure 4.2, which are straightforward for the most part:

The two primitive behaviors **input** and **time** have type B Input and B Real respectively, where Input is a placeholder for the actual input to the system.

The nullary lifting operation **lift** e turns a base language expression e of type α into a behavior of type B α. The unary lifting **lift** $(\lambda x.e)\ b$ builds a behavior of type B α', where b is a B α and $\lambda x.e$ is a function from α to α'. Lifting operations of higher arities are typed similarly.

The **integral** operator only works on Real behaviors, and returns a Real behavior.

Let-behavior defines a mutually recursive group of behaviors, each parameterized by a base language variable. The corresponding typing rule reflects that the behavior variables can occur free in their definitions as well as the body of **let-behavior**.

In b **until** $\langle ev_j \Rightarrow z_j \rangle$, b and ev_j cannot contain free behavior variables.

The predicate event operator **when** turns a behavior of Bool to an event of the unit type.

liftev $(\lambda(x_0, \cdots, x_n).e)\ ev\ b_1\ \cdots b_n$ is an event of type α, where ev is an α_0 event, b_1, ..., b_n are behaviors of type α_1, ..., α_n respectively, and $\lambda(x_0, \cdots, x_n).e$ is a function from

$(\alpha_0, \alpha_1, \cdots, \alpha_n)$ to α.

$$\boxed{\Gamma;\ \Omega \vdash\ b : \mathsf{B}\ \alpha}$$

$$\frac{}{\Gamma;\ \Omega \vdash\ \textbf{input} : \mathsf{B}\ \mathsf{Input}} \qquad \frac{}{\Gamma;\ \Omega \vdash\ \textbf{time} : \mathsf{B}\ \mathsf{Real}}$$

$$\frac{\{\Gamma;\ \Omega \vdash\ b_j : \mathsf{B}\ \alpha_j\} \quad \Gamma \uplus \{x_j : \alpha_j\} \vdash\ e : \alpha}{\Gamma;\ \Omega \vdash\ \textbf{lift}\ (\lambda(x_1, \cdots, x_n).e)\ b_1\ \cdots\ b_n : \mathsf{B}\ \alpha} \qquad \frac{\Gamma;\ \Omega \vdash\ b : \mathsf{B}\ \mathsf{Real}}{\Gamma;\ \Omega \vdash\ \textbf{integral}\ b : \mathsf{B}\ \mathsf{Real}}$$

$$\frac{\{\ \Gamma \oplus \{x_i : \alpha_i\};\ \Omega' \vdash\ w_i : \alpha'_i\ \}^{i \in J} \quad \Gamma;\ \Omega' \vdash\ b : \mathsf{B}\ \alpha \qquad \Omega' \equiv \Omega \oplus \{z_j : \alpha_j \to \alpha'_j\}^{j \in J}}{\Gamma;\ \Omega \vdash\ \textbf{let behavior}\ \{\ z_j(x_j)\ =\ w_j\ \}^{j \in J}\ \textbf{in}\ b : \mathsf{B}\ \alpha}$$

$$\frac{\Gamma;\ \emptyset \vdash\ b : \mathsf{B}\ \alpha \qquad \{\ \Gamma;\ \emptyset \vdash\ ev_j : \mathsf{E}\ \alpha_j\ \}}{\Gamma;\ \Omega \uplus \{z_j : \alpha_j \to \alpha\} \vdash\ b\ \textbf{until}\ \langle\ ev_j\ \Rightarrow\ z_j\ \rangle : \mathsf{B}\ \alpha}$$

$$\boxed{\Gamma;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha}$$

$$\frac{\Gamma;\ \Omega \vdash\ b : \mathsf{B}\ \mathsf{Bool}}{\Gamma;\ \Omega \vdash\ \textbf{when}\ b : \mathsf{E}\ ()} \qquad \frac{\Gamma;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha_0 \quad \{\Gamma;\ \Omega \vdash\ b_j : \mathsf{B}\ \alpha_j\}^{j \in \mathbb{N}_n} \quad \Gamma \oplus \{x_j : \alpha_j\}^{j \in \{0..n\}} \vdash\ e : \alpha}{\Gamma;\ \Omega \vdash\ \textbf{liftev}\ (\lambda(x_0, \cdots, x_n).e)\ ev\ b_1\ \cdots\ b_n : \mathsf{E}\ \alpha}$$

Figure 4.2: Type system of H-FRP

4.3 Denotational semantics

Next, we give a denotational semantics to H-FRP by defining two semantic functions: **at**$[\![-]\!]$ for behaviors and **occ**$[\![-]\!]$ for events. We will also need an auxiliary function **eval**$[\![-]\!]$ to evaluate a base language term.

4.3.1 Semantic functions

To define the semantic functions, we need to introduce the following notations:

- $A_\perp$ (read "A *lifted*") stands for the set obtained by adding the *bottom element* $\perp$ to set A, i.e.

$$A_\perp \stackrel{\text{def}}{=} A \uplus \{\perp\}.$$

- $v_\perp$ is the syntactic category for *optional values*:

$$v_\perp \quad ::= \quad v \;\Big|\; \perp$$

- $\mathcal{B}$ is a *behavior environment*, which maps behavior variables to their definitions:

$$\mathbb{B} \ni \mathcal{B} \quad ::= \quad \{z_j \mapsto \lambda x_j.w_j\}$$

- and $\mathcal{X}_\perp$ is a *lifted variable environment*, which maps variables to optional values:

$$\mathbb{X}_\perp \ni \mathcal{X}_\perp \quad ::= \quad \{x_j \mapsto v_{\perp j}\}$$

The signatures of the semantic functions can then be given as:

$$\begin{array}{rcl} \textbf{at}[\![-]\!] & : & B \to \mathbb{B} \to \mathbb{T} \to \mathbb{T} \to V_\perp \\ \textbf{occ}[\![-]\!] & : & EV \to \mathbb{B} \to \mathbb{T} \to \mathbb{T} \to (\mathsf{Maybe}\ (\mathbb{T}, V))_\perp \\ \textbf{eval}[\![-]\!] & : & E \to \mathbb{X}_\perp \to V_\perp \end{array}$$

The function $\textbf{at}[\![-]\!]$ takes a behavior term b, a behavior environment $\mathcal{B}$ that defines all free behavior variables in b, a start time t_0, and a current time t that satisfies $t \geq t_0$. It returns the value of b (which was started at time t_0) at time t. Note that the result has type $V_\perp$ rather than V. The reason is that, just like the standard practice of using "bottom" ($\perp$) to denote abnormal situations (programs that generate run-time errors or

do not terminate, for example) in denotational semantics, we need something to denote run-time anomaly of an H-FRP program.

However, we must emphasize that one is not to equate the bottom element in H-FRP to that in traditional work on denotational semantics, for there is a major difference between the two: the bottom element ($\bot$) here does *not* denote run-time error or non-termination. Instead, it just means "no information." For example, $\mathbf{at}[\![b]\!]_{\mathcal{B}}\ t_0\ t = \bot$ simply means that the value of behavior b, starting at time t_0, at time t is *unspecified*. This unusual interpretation of $\bot$ might be surprising at first, but since H-FRP is for real-time systems, run-time error or non-termination is not acceptable. In fact, as we will see in Section 4.4, we can execute an H-FRP program on a real computer by first translating it to RT-FRP, which is then guaranteed to be resource-bounded and will never generate run-time errors.

As we have said in Section 4.1, events in H-FRP can have only one occurrence. The function $\mathbf{occ}[\![-]\!]$ gives meaning to an event term ev. Its arguments include ev, a behavior environment $\mathcal{B}$ that defines all free behavior variables in ev, a start time t_0, and a current time t. It returns Nothing if ev hasn't occurred since t_0 up to t, or Just (τ, v) if ev occurred at time $\tau \in [t_0, t)$ with value v. It may also return $\bot$ if there is not enough information to determine an occurrence or the absence of it.

Finally, the auxiliary function $\mathbf{eval}[\![-]\!]$ evaluates a base language term e. It takes a lifted environment $\mathcal{X}_{\perp\!\!\perp}$, which may map a variable to $\bot$, strips $\mathcal{X}_{\perp\!\!\perp}$ of those mappings leading to $\bot$, and then evaluates e in this stripped-down environment $\mathcal{X}$. If $\mathcal{X}$ does not contain enough information to evaluate e (for example, when $\mathcal{X}_{\perp\!\!\perp}$ maps a free variable in e to $\bot$), $\bot$ will be returned.

$$\boxed{\mathbf{eval}[\![e]\!]x_{\perp\!\!\perp}}\ \mathbf{eval}[\![e]\!]x_{\perp\!\!\perp} = \begin{cases} v & \exists v.\ \mathcal{X} \vdash e \hookrightarrow v \text{ where } \mathcal{X}_{\perp\!\!\perp} \equiv \mathcal{X} \uplus \{x_k \mapsto \bot\} \\ & \quad \text{and } \mathcal{X} \equiv \{x_j \mapsto v_j\} \\ \bot & \text{otherwise} \end{cases}$$

4.3.2 Environment-context compatibility

Naturally, when studying the semantics of a behavior term b in environment $\mathcal{B}$, we would only use a $\mathcal{B}$ that "makes sense" to b. For example, the term

$$b \equiv (\textbf{lift}\ 0)\ \textbf{until}\ \langle ev \Rightarrow \mathsf{z} \rangle$$

is meaningless in the environment

$$\mathcal{B} \equiv \{\mathsf{z} \mapsto \lambda \mathsf{x}.\textbf{lift}\ \mathsf{False}\},$$

because it requires b to change from a numeric behavior to a Boolean behavior when ev occurs.

To assert that such a run-time type error never occurs, we need the following concept:

Definition 4.1 (Environment-context compatibility). A behavior environment $\mathcal{B} \equiv \{z_j \mapsto \lambda x_j.w_j\}^{j \in \mathbb{N}_n}$ is *compatible with* behavior context Ω, if $\Omega \equiv \Omega' \uplus \{z_j : \alpha_j \to \alpha'_j\}^{j \in \mathbb{N}_n}$, and $\forall j \in \mathbb{N}_n.\ \{x_j : \alpha_j\}; \Omega \vdash w_j : \mathsf{B}\ \alpha'_j$. We denote this symbolically by

$$\Omega \vdash \mathcal{B}.$$

The intuition behind this concept is not difficult. A behavior context Ω assigns types to behavior variables, and a behavior environment $\mathcal{B}$ defines behavior variables. When we say $\mathcal{B}$ is compatible with Ω, we expect for all z defined in $\mathcal{B}$, it has a type in Ω, and its definition in $\mathcal{B}$ has the same type.

Note that since we are only interested in behavior environments where all definitions are closed, we do not need to talk about a variable context Γ when we discuss the compatibility of $\mathcal{B}$ with Ω.

4.3.3 Definition of at$[\![-]\!]$ and occ$[\![-]\!]$

In defining the semantics of H-FRP, and in the rest of the chapter, we assume that there is a global function named "input", which has type $\mathbb{T} \to$ Input and specifies the input to the system at any given time. Hence, the meaning of a program is parameterized over this function.

The denotational semantics of well-typed closed H-FRP terms is given in Figure 4.3 and Figure 4.4, where we require $t \geq t_0$, and $\int$ is the Riemann integral extended to functions that might return $\perp$ at a *finite* number of points:

$$\int_{t_0}^{t} f(\tau)\mathrm{d}\tau = \begin{cases} \int_{t_0}^{t} f'(\tau)\mathrm{d}\tau & f(\tau) \neq \perp \text{ on } [t_0, t] \text{ except a finite set } S \subset \mathbb{T}, \\ & f'(\tau) \stackrel{\text{def}}{=} \text{if } f(\tau) = \perp \text{ then } 0 \text{ else } f(\tau), \\ \perp & \text{otherwise} \end{cases}$$

and the map function is defined as:

$$\begin{aligned} \mathrm{map} &: (\alpha \to \alpha') \to (\mathsf{Maybe}\ \alpha)_\perp \to (\mathsf{Maybe}\ \alpha')_\perp \\ \mathrm{map}\ f\ \perp &= \perp \\ \mathrm{map}\ f\ \mathsf{Nothing} &= \mathsf{Nothing} \\ \mathrm{map}\ f\ (\mathsf{Just}\ v) &= \mathsf{Just}\ (f\ v) \end{aligned}$$

The semantics of **when** is particularly involved. To help the reader gain an intuition, we illustrate it in Figure 4.5, 4.6, and 4.7. In particular:

1. Figure 4.5 shows the general case and two special cases where **when** b has no occurrence on $[t_0, t)$. A solid round end of a line segment emphasizes that the value of the behavior b at that point is significant, while a normal end means that the value of b at that end is irrelevant. For example, part (a) of the figure should be interpreted as: b is True on $[t_0, T_1)$, False on (T_1, t), and we do not care about b's value at T_1 and t; part (b) means: b is False on (t_0, t), and its value at t_0 or t is not important; and so

$\boxed{\mathbf{at}[\![b]\!]_{\mathcal{B}}}$

$$\mathbf{at}[\![\mathbf{input}]\!]_{\mathcal{B}}\ t_0\ t = \mathrm{input}(t)$$

$$\mathbf{at}[\![\mathbf{time}]\!]_{\mathcal{B}}\ t_0\ t = t$$

$$\mathbf{at}[\![\mathbf{lift}\ (\lambda(x_1,\cdots,x_n).e)\ b_1\ \cdots\ b_n]\!]_{\mathcal{B}}\ t_0\ t = \mathbf{eval}[\![e]\!]_{\{x_j \mapsto \mathbf{at}[\![b_j]\!]_{\mathcal{B}}\ t_0\ t\}}$$

$$\mathbf{at}[\![\mathbf{integral}\ b]\!]_{\mathcal{B}}\ t_0\ t = \int_{t_0}^{t} (\mathbf{at}[\![b]\!]_{\mathcal{B}}\ t_0\ \tau)\,\mathrm{d}\tau$$

$$\mathbf{at}[\![\mathbf{let\ behavior}\ \{z_j(x_j) = w_j\}\ \mathbf{in}\ b]\!]_{\mathcal{B}}\ t_0\ t = \mathbf{at}[\![b]\!]_{\mathcal{B}\cup\{z_j \mapsto \lambda x_j.w_j\}}\ t_0\ t$$

$$\mathbf{at}[\![b\ \mathbf{until}\ \langle ev_j \Rightarrow z_j\rangle^{j\in\mathbb{N}_n}]\!]_{\mathcal{B}}\ t_0\ t$$

$$= \begin{cases} \mathbf{at}[\![b]\!]_{\mathcal{B}}\ t_0\ t & \forall j \in \mathbb{N}_n, o_j(t) = \mathsf{Nothing} \\ \mathbf{at}[\![w[x := v]]\!]_{\mathcal{B}}\ \tau\ t & \exists T \in (t_0, t].\ o_k(T) = \mathsf{Just}\ (\tau, v),\ \lambda x.w = \mathcal{B}(z_k), \\ & \quad \text{and } \forall j \in \mathbb{N}_n - \{k\}, o_j(T) = \mathsf{Nothing} \\ \bot & \text{otherwise} \end{cases}$$

where $o_j(t) = \mathbf{occ}[\![ev_j]\!]_{\mathcal{B}}\ t_0\ t$

Figure 4.3: Denotational semantics of behaviors

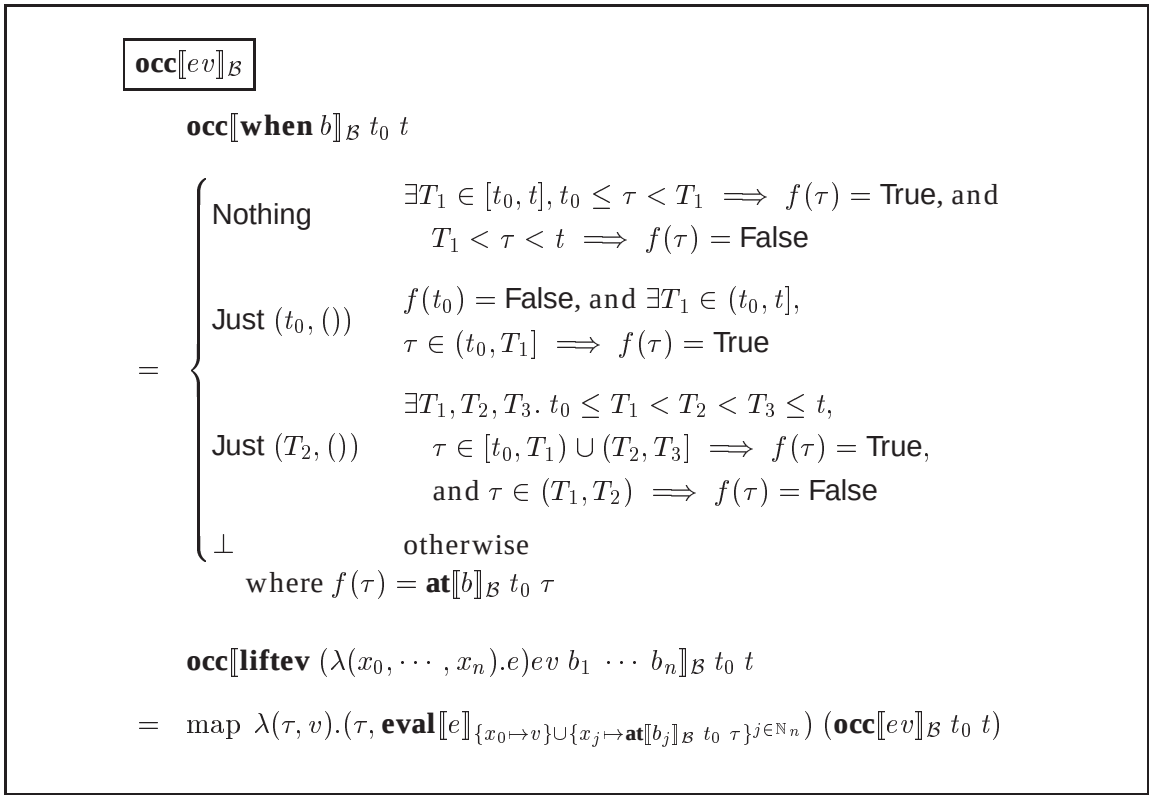


Figure 4.4: Denotational semantics of events

on.

2. Figure 4.6 shows the cases where **when** b occurs on $[t_0, t)$. An upward arrow is used to indicate the time of the occurrence, and a hollow circle in a line segment means the value of b at that point is *not* the same as that at the other points in the segment. For example, part (a) of the figure means that **when** b occurs at time t_0 if b is False at t_0 and True on (t_0, T_1).

3. Figure 4.7 illustrates that when there is a glitch in b, either negative or positive, we do not know whether **when** b occurs on $[t_0, t)$.

Finally, we point out that H-FRP is type preserving in the following sense:

Lemma 4.1 (Type preservation of semantics). Given any $b \in B$, $ev \in EV$, $\mathcal{B} \in \mathbb{B}$, and type α, we have

1. $\emptyset;\ \Omega \vdash\ b : \mathsf{B}\ \alpha$ and $\Omega \vdash \mathcal{B} \implies \forall t_0, t \in \mathbb{T}.\ \mathbf{at}[\![b]\!]_{\mathcal{B}}\ t_0\ t \in \alpha_\perp$, and

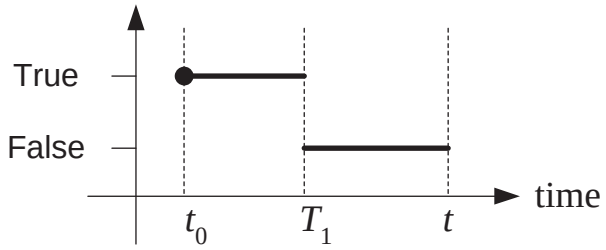
2. $\emptyset;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha$ and $\Omega \vdash \mathcal{B} \implies \forall t_0, t \in \mathbb{T}.\ \mathbf{occ}[\![ev]\!]_{\mathcal{B}}\ t_0\ t \in (\mathsf{Maybe}\ (\mathbb{T}, \alpha))_\perp$.

Proof. By co-induction on the derivation trees for $\emptyset;\ \Omega \vdash\ b : \mathsf{B}\ \alpha$ and $\emptyset;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha$. □

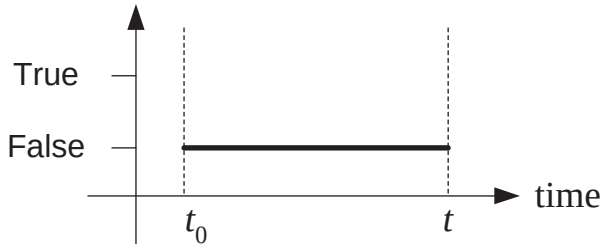
4.4 Operational semantics via translation to RT-FRP

Although we now have a denotational semantics for H-FRP, we cannot derive an implementation from it yet. The reason is that the semantics uses some continuous operations not available for digital computers, for example, Riemann integral and search over a continuum of real numbers (the latter is used in the semantics of the **until** construct).

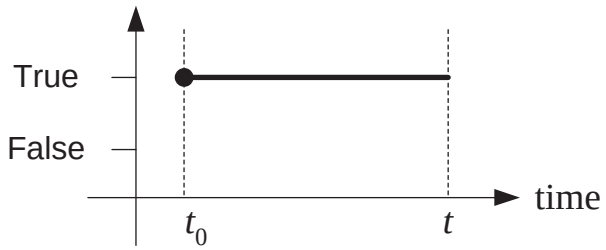
To guide the implementation, we give an operational semantics to H-FRP in this section. Although we could come up with a direct operational semantics, we choose to instead present a translation from H-FRP to RT-FRP, and then use the operational semantics of RT-FRP to execute the target program. We have two reasons for doing this:



(a). general case ($t_0 \leq T_1 \leq t$)

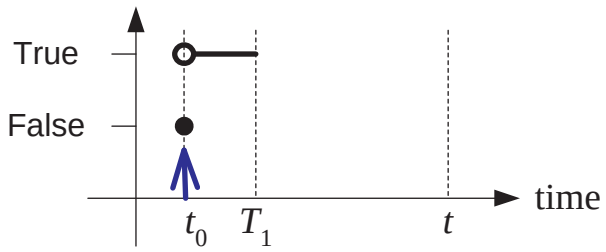


(b). $t_0 = T_1$

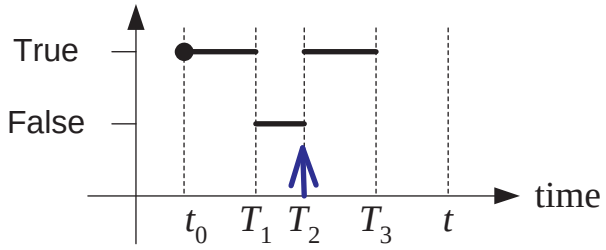


(c). $T_1 = t$

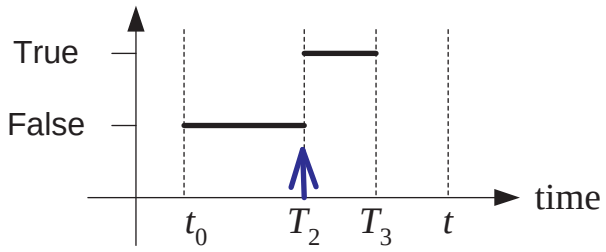
Figure 4.5: Semantics of **when**: no occurrence



(a). Immediate occurrence

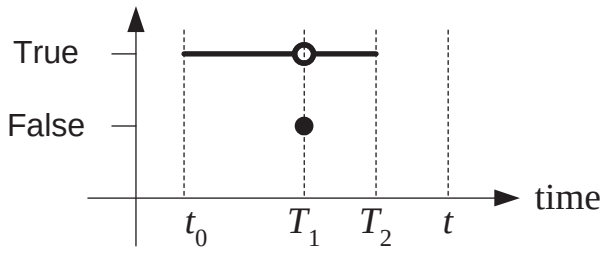


(b). Late occurrence: general case

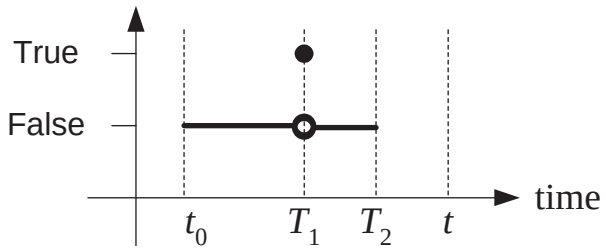


(c). Late occurrence: special case ($t_0 = T_1$)

Figure 4.6: Semantics of **when**: occurrence



(a). Negative glitch ($t_0 < T_1 < T_2 \leq t$)



(b). Positive glitch ($t_0 < T_1 < T_2 \leq t$)

Figure 4.7: Semantics of **when**: not enough information

1. This approach involves less work than defining a direct operational semantics; and

2. By having such a translation, we clearly show that H-FRP can be regarded as a subset of RT-FRP.

Figure 4.8 defines the function $\mathbf{tr}[\![-]\!]$ that translates H-FRP terms to RT-FRP. The definition is syntax-directed.

Please note that, since we require that all behavior variables have distinct names in the source program, and the translation only introduces fresh names, the **let-signal** clauses obtained from translating **lift** are not recursive, i.e. the variables $\{x_j\}$ cannot occur in the terms $\{\mathbf{tr}[\![b_j]\!]\}$.

Naturally, we expect that the translation function $\mathbf{tr}[\![-]\!]$ turns a well-typed H-FRP program into a well-typed RT-FRP program. In addition, we expect that an H-FRP behavior of type α is translated to a signal of the same type, and an H-FRP event of type α is translated to a signal of type $\mathsf{Maybe}\ \alpha$. This property is formalized by the following lemma:

Lemma 4.2 (Type preservation of translation). For all variable contexts Γ and Θ, signal context Ω, and terms b and ev, we have

1. $\Gamma;\ \Omega \vdash\ b : \mathsf{B}\ \alpha \implies \Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ \mathbf{tr}[\![b]\!] : \alpha$, and

2. $\Gamma;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha \implies \Gamma;\ \Theta;\ \Delta \vdash_{\mathbf{S}}\ \mathbf{tr}[\![ev]\!] : \mathsf{Maybe}\ \alpha$.

Proof. The proof of this lemma is a co-induction on the derivation trees of $\Gamma;\ \Omega \vdash\ b : \mathsf{B}\ \alpha$ and $\Gamma;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha$. □

4.5 Convergence of semantics

We have shown that an H-FRP program can be translated to an RT-FRP program, which can then be executed in discrete steps using the operational semantics we defined in Section 3.2.

tr⟦b⟧

$$\begin{aligned}
\textbf{tr}[\![\textbf{input}]\!] &= \textbf{input} \\
\textbf{tr}[\![\textbf{time}]\!] &= \textbf{time} \\
\textbf{tr}[\![\textbf{lift}\ (\lambda(x_1,\cdots,x_n).e)\ b_1\ \cdots\ b_n]\!] &= \textbf{let snapshot}\ x_1 \leftarrow \textbf{tr}[\![b_1]\!]\ \textbf{in} \\
&\quad\ \vdots \\
&\quad \textbf{let snapshot}\ x_n \leftarrow \textbf{tr}[\![b_n]\!]\ \textbf{in} \\
&\quad\ \textbf{ext}\ e \\
\textbf{tr}[\![\textbf{integral}\ b]\!] &= \textbf{let snapshot}\ t \leftarrow \textbf{time}\ \textbf{in} \\
&\quad \textbf{let snapshot}\ x \leftarrow \textbf{tr}[\![b]\!]\ \textbf{in} \\
&\quad\ \textbf{let snapshot}\ (i_0, x_0, t_0) \leftarrow \\
&\quad\quad \textbf{delay}\ (0,0,0) \\
&\quad\quad\ \ \textbf{ext}\ (i_0 + x_0 \cdot (t - t_0), x, t) \\
&\quad\ \textbf{in ext}\ i_0 \\
\textbf{tr}[\![\textbf{let behavior}\ \{z_j(x_j) = w_j\}\ \textbf{in}\ b]\!] &= \textbf{let signal}\ \{z_j(x_j) = \textbf{tr}[\![w_j]\!]\}\ \textbf{in}\ \textbf{tr}[\![b]\!] \\
\textbf{tr}[\![b\ \textbf{until}\ \langle ev_j \Rightarrow z_j \rangle]\!] &= \textbf{tr}[\![b]\!]\ \textbf{until}\ \langle \textbf{tr}[\![ev_j]\!] \Rightarrow z_j \rangle
\end{aligned}$$

tr⟦ev⟧

$$\begin{aligned}
\textbf{tr}[\![\textbf{when}\ b]\!] &= \textbf{let snapshot}\ x \leftarrow \textbf{tr}[\![b]\!]\ \textbf{in} \\
&\quad \textbf{let snapshot}\ x_0 \leftarrow \\
&\quad\quad \textbf{delay}\ \mathsf{True}\ (\textbf{ext}\ x) \\
&\quad\ \textbf{in ext}\ (\neg x_0 \wedge x) \\
\textbf{tr}[\![\textbf{liftev}\ (\lambda(x_0,\cdots,x_n).e)\ ev\ b_1\ \cdots\ b_n]\!] &= \textbf{let snapshot}\ x \leftarrow \textbf{tr}[\![ev]\!]\ \textbf{in} \\
&\quad \textbf{let snapshot}\ x_1 \leftarrow \textbf{tr}[\![b_1]\!]\ \textbf{in} \\
&\quad\ \vdots \\
&\quad \textbf{let snapshot}\ x_n \leftarrow \textbf{tr}[\![b_n]\!]\ \textbf{in} \\
&\quad\ \textbf{ext}\ (\textbf{case}\ x\ \textbf{of}\ \mathsf{Just}\ x_0 \Rightarrow \mathsf{Just}\ e \\
&\quad\quad\ \ \textbf{else}\ \mathsf{Nothing})
\end{aligned}$$

tr⟦Ω⟧

$$\textbf{tr}[\![\{z_j \mapsto \lambda x_j.w_j\}]\!] = \{z_j \mapsto \lambda x_j.\textbf{tr}[\![w_j]\!]\}$$

Figure 4.8: Translation from H-FRP to RT-FRP

Normally, when we give semantics to a language in more than one way, we expect those semantics to "agree with each other" in the sense that they will always predict the same result given the same input to the program. However, this is not the case for H-FRP, as its denotational semantics implies we can always update the system state in infinitesimal time, while its operational semantics says the state is only updated in discrete, countable, steps. Therefore, the denotational semantics describes the idealized situation where we have infinite computational power.

Nonetheless, we expect some correspondence between the two semantics of H-FRP: namely, when we execute a program using the operational semantics on an infinitely fast imaginary computer, where the time between two execution steps is infinitesimal, we expect the result to match that given by the denotational semantics. The rest of this section establishes this relation under certain conditions.

4.5.1 Definitions and concepts

To formalize the relation between the two semantics, we need to define some concepts.

Time sequences and functions on them

First, we define *TS* as the set of *time sequences,* which are non-empty sequences of times in ascending order, i.e.

$$TS \stackrel{\text{def}}{=} \{\langle t_1, \cdots, t_n\rangle \mid n \in \mathbb{N}, \{t_1, \cdots, t_n\} \subset \mathbb{T}, \text{ and } t_1 < \cdots < t_n\}.$$

Given a time sequence $P \equiv \langle t_1, \cdots, t_n\rangle$, we often write it as P^t where $t = t_n$ to emphasize that the last time in P is t.

The auxiliary function $\widetilde{\text{trace}}[\![-]\!]$ will be used to observe the sequence of values gener-

ated by executing an H-FRP term:

$$\widetilde{\text{trace}}[\![-]\!] : (B \uplus EV) \to \mathbb{B} \to \mathit{TS} \to \langle V \rangle$$
$$\widetilde{\text{trace}}[\![E]\!]_{\mathcal{B}}\ \langle t_1, \cdots, t_n \rangle \stackrel{\text{def}}{=} \langle v_1, \cdots, v_n \rangle$$
$$\text{where } \emptyset;\ \mathbf{tr}[\![\mathcal{B}]\!] \vdash \mathbf{tr}[\![E]\!] \overset{t_1, i_1}{\hookrightarrow\!\!\!\!\rightarrow} v_1, s_1 \overset{t_2, i_2}{\hookrightarrow\!\!\!\!\rightarrow} \cdots \overset{t_n, i_n}{\hookrightarrow\!\!\!\!\rightarrow} v_n, s_n \text{ and } \forall j.\ i_j = \text{input}(t_j).$$

Next, we define a function $\widetilde{\text{at}}[\![-]\!]$, which gives the last value yielded by executing a behavior for n steps:

$$\widetilde{\text{at}}[\![-]\!] : B \to \mathbb{B} \to \mathit{TS} \to V$$
$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ ts \stackrel{\text{def}}{=} v_n \text{ where } \widetilde{\text{trace}}[\![b]\!]_{\mathcal{B}}\ ts \equiv \langle v_1, \cdots, v_n \rangle$$

and a function $\widetilde{\text{occ}}[\![-]\!]$, which returns the possible occurrence of an event in its first n steps of execution:

$$\widetilde{\text{occ}}[\![-]\!] : EV \to \mathbb{B} \to \mathit{TS} \to \mathsf{Maybe}\ (\mathbb{T}, V)$$
$$\widetilde{\text{occ}}[\![ev]\!]_{\mathcal{B}}\ ts \stackrel{\text{def}}{=} \begin{cases} \mathsf{Nothing} & \forall j \in \mathbb{N}_{n-1}, v_j \equiv \mathsf{Nothing} \\ \mathsf{Just}\ (t_k, v) & k \in \mathbb{N}_{n-1}, v_k \equiv \mathsf{Just}\ v, \text{ and } \forall j \in \mathbb{N}_{k-1}, v_j \equiv \mathsf{Nothing} \end{cases}$$
$$\text{where } \widetilde{\text{trace}}[\![ev]\!]_{\mathcal{B}}\ ts \equiv \langle v_1, \cdots, v_n \rangle.$$

We would like to prove that, as the sampling rate approaches infinity, $\widetilde{\text{at}}[\![-]\!]$ and $\widetilde{\text{occ}}[\![-]\!]$ will eventually agree with $\mathbf{at}[\![-]\!]$ and $\mathbf{occ}[\![-]\!]$, respectively. To establish such a connection, we first need to formally define what it means by "the sampling rate approaches infinity". In the rest of the chapter, we will use the notation Δt_j for $t_{j+1} - t_j$. [2]

Definition 4.2 (Norm of time sequence). Given a time sequence $P \equiv \langle t_1, \cdots, t_n \rangle \in \mathit{TS}$, the *norm* of P, written $|P|$, is defined as the maximum of the set $\{0\} \cup \{\Delta t_j\}^{j \in \mathbb{N}_{n-1}}$. The *norm* of P with respect to start time t_0, written $|P|_{t_0}$, is defined as $\max\{|\Delta t_0|, |P|\}$.

[2] The symbol Δ must not be interpreted as a factor but only as indicating a difference in values of the variable which follows.

A metric on values

To study the limit of the operational semantics as the sampling interval goes to zero, we need a way to measure how close two values are to each other, and we must consider values of all types, not just real numbers. Hence, we define a *metric* on values as:

Definition 4.3 (Difference between values). The *difference* between two values v and v', of the same type, written $|\, v - v' \,|$, is the real number defined by

$$
\begin{array}{rcl}
|\, () - () \,| & = & 0 \\
|\, r - r' \,| & = & \text{the absolute value of } r - r', \text{ where} \\
 & & \quad \text{“}-\text{” is real number subtraction} \\
|\, (v_1, \cdots, v_n) - (v'_1, \cdots, v'_n) \,| & = & \Sigma_{j=1}^{n} \left|\, v_j - v'_j \,\right| \\
|\, \mathsf{Nothing} - \mathsf{Nothing} \,| & = & 0 \\
|\, \mathsf{Just}\ v - \mathsf{Just}\ v' \,| & = & |\, v - v' \,| \\
|\, \mathsf{Nothing} - \mathsf{Just}\ v \,| & = & 1 \\
|\, \mathsf{Just}\ v - \mathsf{Nothing} \,| & = & 1
\end{array}
$$

It is obvious that $|\, v - v' \,| = |\, v' - v \,|$ and $|\, v - v \,| = 0$.

Limit of operational semantics

Now that we have a metric on all values, we can define the limit of the operational semantics when the sampling interval goes to zero as:

$$\widetilde{\text{at}}^*[\![-]\!] : B \to \mathbb{B} \to \mathbb{T} \to \mathbb{T} \to V_\perp$$

$$\widetilde{\text{at}}^*[\![b]\!]_{\mathcal{B}}\ t_0\ t \stackrel{\text{def}}{=} \begin{cases} \lim_{|P^t|_{t_0} \to 0} \widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t & \text{if such limit exists} \\ \perp & \text{otherwise} \end{cases}$$

$$\widetilde{\text{occ}}^*[\![-]\!] : EV \to \mathbb{B} \to \mathbb{T} \to \mathbb{T} \to (\mathsf{Maybe}\ (\mathbb{T}, V))_\perp$$

$$\widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ t \stackrel{\text{def}}{=} \begin{cases} \lim_{|P^t|_{t_0} \to 0} \widetilde{\text{occ}}[\![ev]\!]_{\mathcal{B}}\ P^t & \text{if such limit exists} \\ \perp & \text{otherwise} \end{cases}$$

Now we can formally ask the question whether the discrete interpretation of an H-FRP program is "faithful" to its continuous interpretation: do the following equations hold?

$$\begin{aligned} \widetilde{\text{at}}^*[\![b]\!]_{\mathcal{B}}\ t_0\ t &= \mathbf{at}[\![b]\!]_{\mathcal{B}}\ t_0\ t \\ \widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ t &= \mathbf{occ}[\![ev]\!]_{\mathcal{B}}\ t_0\ t \end{aligned}$$

Or more concisely, are the following true?

$$\begin{aligned} \widetilde{\text{at}}^*[\![-]\!] &= \mathbf{at}[\![-]\!] \\ \widetilde{\text{occ}}^*[\![-]\!] &= \mathbf{occ}[\![-]\!] \end{aligned}$$

The following section establishes this result. However, as we will see, such correspondence is conditional. We will present a set of conditions that are sufficient to guarantee the correspondence. Such conditions will be made exact through a series of theorems presented in this section. We will prove that for programs satisfying this set of conditions, the limit of the operational semantics does converge to the denotational semantics, and will show examples of programs that do not meet such conditions and therefore whose operational semantics do not converge to the denotational semantics.

Convergence and uniform convergence

We will need the following concepts in our theorems:

Definition 4.4 (ϵ-neighborhood). Given a real number $\epsilon > 0$, the *ϵ-neighborhood* of a value v, written $[v]_\epsilon$, is the set of values whose difference to v is less than ϵ:

$$[v]_\epsilon \stackrel{\text{def}}{=} \{x \mid |\, x - v \,| < \epsilon\}.$$

The *ϵ-neighborhood* of a set of values S, written $[S]_\epsilon$, is the union of the ϵ-neighborhoods of the elements of S:

$$[S]_\epsilon \stackrel{\text{def}}{=} \bigcup_{v \in S} [v]_\epsilon.$$

Definition 4.5 (Uniform convergence). Given times t_0 and T where $T \geq t_0$, a set of times S, and two functions $F : TS \to \alpha$, and $f : \mathbb{T} \to \alpha$, we say that

$$F \textit{ uniformly converges to } f \textit{ on } [t_0, T] \textit{ except near } S,$$

if for every $\epsilon > 0$, there exists a $\delta > 0$ (depending only on ϵ) such that for any time sequence P^t where $t \leq T$, $t \notin [S]_\epsilon$, and $\left|P^t\right|_{t_0} < \delta$, we have

$$\left|\, F(P^t) - f(t) \,\right| < \epsilon.$$

We denote this symbolically by writing

$$F(P^t) \rightarrowtail f(t)\Big|_{t_0}^{T} - S.$$

We omit "$-S$" when $S = \emptyset$.

Although the definition of uniform convergence may seem daunting, the intuition is simple: when we say a function on time sequences uniformly converges, we mean that no matter how small the error bound we demand, we can always find a threshold δ, such

that as long as the sampling interval is smaller than δ, the result at every sample point is guaranteed to be within the error bound.

In the above definition, we require that P^t satisfies that $t \leq T$, $t \notin [S]_\epsilon$, and $\left|P^t\right|_{t_0} < \delta$. This condition can be rephrased more verbosely (but perhaps more understandably) as $t_1 \in [t_0]_\delta$, $t_1 \leq t \leq T$, $t \notin [S]_\epsilon$, and $\left|P^t\right| < \delta$, where $P^t = \langle t_1, \cdots, t \rangle$.

Why are we interested in uniform convergence rather than plain convergence? After all, why should we care about whether the convergence is uniform or not? To understand our decision, let's study a program called $bizarre$, inspired by [3, page 401]:

$$
\begin{aligned}
bizarre \equiv\ & \textbf{let}\ \mathsf{z(t1)} = (\textbf{lift}\ (\lambda \mathsf{t}.\textbf{let}\ \mathsf{c} = 1/\mathsf{t1}\ \textbf{in}\ \mathsf{c} \cdot \mathsf{c} \cdot \mathsf{t} \cdot \mathsf{pow}(1-\mathsf{t},\mathsf{c}))\ \textbf{time}) \\
& \qquad \textbf{until}\ \langle\rangle \\
& \textbf{in}\ (\textbf{lift}\ 0)\ \textbf{until}\ \langle \textbf{liftev}\ (\lambda(\_,\mathsf{t}).\mathsf{t}) \\
& \qquad\qquad (\textbf{when}\ (\textbf{lift}\ (\lambda \mathsf{t}.\mathsf{t} > 0)\ \textbf{time})) \\
& \qquad\qquad \textbf{time} \\
& \qquad \Rightarrow \mathsf{z} \rangle
\end{aligned}
$$

Following the operational semantics of H-FRP, it is not difficult to see that

$$
\begin{aligned}
& \widetilde{\mathsf{trace}}[\![bizarre]\!]_{\mathcal{B}}\ \langle t_1, \cdots, t_n \rangle \\
= & \begin{cases} \langle 0, 0, v_3, v_4, \cdots, v_n \rangle & t_1 = 0, \text{ where } v_j = c^2 t (1-t)^c \text{ and } c = 1/t_2 \\ \langle 0, v_2, v_3, \cdots, v_n \rangle & t_1 > 0, \text{ where } v_j = c^2 t (1-t)^c \text{ and } c = 1/t_1 \end{cases}
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
& \widetilde{\mathsf{at}}^*[\![bizarre]\!]_{\mathcal{B}}\ 0\ 0 \\
= & \lim_{|P^0|_0 \to 0} (\widetilde{\mathsf{at}}[\![bizarre]\!]_{\mathcal{B}}\ P^0) \\
= & \lim_{|P^0|_0 \to 0} 0 \\
= & 0
\end{aligned}
$$

and for any $t \in (0, 1]$, we have

$$\begin{aligned}
& \widetilde{\text{at}}^*[\![bizarre]\!]_{\mathcal{B}}\ 0\ t \\
= \ & \lim_{|P^t|_0 \to 0}(\widetilde{\text{at}}[\![bizarre]\!]_{\mathcal{B}}\ P^t) \\
= \ & \lim_{|P^t|_0 \to 0}(c^2 t(1-t)^c \text{ where } P^t \equiv \langle t_1, \cdots, t_n \rangle, \text{ and} \\
& \qquad T = \text{if } t_1 > 0 \text{ then } t_1 \text{ else } t_2, \text{ and } c = 1/T) \\
= \ & \lim_{T \to 0}(c^2 t(1-t)^c \text{ where } c = 1/T) \\
= \ & 0
\end{aligned}$$

Hence, we have that for all $t \in [0, 1]$, $\widetilde{\text{at}}^*[\![bizarre]\!]_{\mathcal{B}}\ 0\ t = 0$, and therefore

$$\int_0^1 (\widetilde{\text{at}}^*[\![bizarre]\!]_{\mathcal{B}}\ 0\ t)\mathrm{d}t = 0.$$

However,

$$\begin{aligned}
& \widetilde{\text{at}}^*[\![\textbf{integral}\ bizarre]\!]_{\mathcal{B}}\ 0\ 1 \\
= \ & \lim_{|P^1|_0 \to 0} \Sigma_{i=1}^{n-1} v_i(t_{i+1} - t_i) \\
& \qquad \text{where } P^1 \equiv \langle t_1, \cdots, t_n \equiv 1 \rangle \text{ and } \widetilde{\text{trace}}[\![bizarre]\!]_{\mathcal{B}}\ P^1 \equiv \langle v_1, \cdots, v_n \rangle \\
= \ & \lim_{|P^1|_0 \to 0} \Sigma_{i=k+1}^{n-1} c^2 t_i(1-t_i)^c(t_{i+1} - t_i) \qquad (*) \\
& \qquad \text{where } k = \text{if } t_1 > 0 \text{ then } 1 \text{ else } 2 \text{ and } c = 1/t_k
\end{aligned}$$

Since

$$\lim_{c \to +\infty} \int_0^1 c^2 t(1-t)^c \mathrm{d}t = \lim_{c \to +\infty} \frac{c^2}{(c+1)(c+2)} = 1,$$

we have

$$(*) = 1.$$

Therefore,

$$\widetilde{\text{at}}^*[\![\textbf{integral}\ bizarre]\!]_{\mathcal{B}}\ 0\ 1 = 1 \neq 0 = \int_0^1 (\widetilde{\text{at}}^*[\![bizarre]\!]_{\mathcal{B}}\ 0\ t)\mathrm{d}t.$$

In other words, as the sampling interval approaches 0, the value of $bizarre$ at every point in $[0, 1]$ eventually goes to 0. Yet, at the same time, the area of the region under the curve of $bizarre$ between time 0 and 1 becomes 1. This is indeed counter-intuitive, and shows that one can not purely rely on his intuition when dealing with functions on real numbers, and that the convergence property of H-FRP programs is trickier than it may look.

This example shows that the fact that the operational semantics of a behavior b converges to some function f does not guarantee that the operational semantics of **integral** b converges to $\int f$. However, as we will prove in Theorem 4.5, if the operational semantics of b *uniformly* converges to f, then the operational semantics of **integral** b will also uniformly converge to $\int f$.

We show that the concept of uniform convergence enjoys the following form of weakening:

Lemma 4.3 (Weakening of uniform convergence). If $F(P^t) \rightarrowtail f(t)\big|_{t_0}^{T} - S$, then for all $T' \leq T$ and $S' \supseteq S$, we have

$$F(P^t) \rightarrowtail f(t)\big|_{t_0}^{T'} - S'.$$

Proof. By the definition of uniform convergence. □

Continuity and uniform continuity

To study the convergence property of **lift**, we will need a concept called *uniform continuity* [3, page 74], as found in most treatments of calculus and analysis:

Definition 4.6 (Uniform continuity). A function f is said to be *uniformly continuous* on a set S if for every real number $\epsilon > 0$, there exists a real number $\delta > 0$ (depending only on ϵ) such that $x, y \in S$ and $|\, x - y \,| < \delta$ imply $|\, f(x) - f(y) \,| < \epsilon$.

Let's compare this concept with that of (non-uniform) *continuity*:

Definition 4.7 (Continuity). A function f defined on S is said to be *continuous* at point $x \in S$, if for every real number $\epsilon > 0$, there exists a real number $\delta > 0$ (depending only on

ϵ) such that $y \in S$ and $|\, x - y \,| < \delta$ imply $|\, f(x) - f(y) \,| < \epsilon$.

We say that f is *continuous* on a set S, if f is continuous at every point in S.

The difference between uniform continuity and continuity may be clarified by the following analogy from [35, page 348]:

1. A man is *literate* if there is a language that he can read and write.

2. A group of men is *literate* if each of its members is literate.

3. A group of men is *uniformly literate* if there is one language that every member of the group can read and write.

Here it is obvious that *uniform literacy* is a property not of the individuals in a group but of the group as a whole; if each of the members of the group is literate, then it follows that the group is literate, but not that the group is uniformly literate.

As an example of continuous but not uniformly continuous functions, $f(t) = 1/t$ is continuous on $(0, 1]$, but not uniformly continuous on the same interval.

Please note that being uniformly continuous does not necessarily mean that the derivative is bounded. For example, $f(t) = \sqrt{1 - t^2}$ is uniformly continuous on $[0, 1]$, but

$$\frac{\mathrm{d}f(t)}{\mathrm{d}t} = \frac{-\,t}{\sqrt{1 - t^2}}$$

and therefore

$$\lim_{t \to 1^-} \frac{\mathrm{d}f(t)}{\mathrm{d}t} = -\infty,$$

which is unbounded.

However, in the case when the domain of the function is a *closed interval*, continuity does imply uniform continuity. The *uniform continuity theorem* is proved in [35, page 349].

Theorem 4.1 (Uniform continuity). If f is continuous on $[a, b]$, then f is uniformly continuous on $[a, b]$.

Note that in Definition 4.6, the function f does not need to be from $\mathbb{R}$ to $\mathbb{R}$. We only require that there is a metric that tells us the difference between two values in the domain (and the range) of f.

Compatibility

Finally, given a well-typed and closed term b, when we study its denotational semantics **at** $[\![b]\!]_{\mathcal{B}}$ we often need to assert that b is well-typed, closed, and executed in a compatible environment $\mathcal{B}$, i.e. there are Ω and α such that

1. $\emptyset;\ \Omega \vdash\ b : \mathsf{B}\ \alpha$, and
2. $\Omega \vdash \mathcal{B}$.

We will write

$$\vdash b; \mathcal{B}$$

to denote this concisely.

Similarly, we write

$$\vdash ev; \mathcal{B}$$

to denote that there are Ω and α such that

1. $\emptyset;\ \Omega \vdash\ ev : \mathsf{E}\ \alpha$, and
2. $\Omega \vdash \mathcal{B}$.

4.5.2 Establishing the convergence property

In the rest of this chapter, we assume that we can represent real numbers and carry out real arithmetic *exactly* in the base language, and the sampling interval can be made arbitrarily small. Despite existing work on computing certain operations on exact representations of real numbers [13, 51], in general our assumption is not true, and we can only approximate real numbers and functions on them. However, as Strachey pointed out [47]: "the

discrepancies are generally small and we usually start by ignoring them. It is only after we have devised a program which would be correct if the functions used were the exact mathematical ones that we start investigating the errors caused by the finite nature of our computer."

Our work addresses the first step suggested by Strachey (finding the values assuming the operations are exact) by giving a continuous-time-based denotational semantics. We also prove that under suitable conditions and assuming exact real operations, the errors in an implementation that uses sampling to approximate continuous time approach zero when the sampling becomes frequent enough. Therefore we also accomplish *part of* the second step (investigating the errors in an implementation using approximation techniques).

Our work is an idealized theory which provides a theoretical basis for the study of hybrid programming languages like H-FRP. What remains to be studied is the quantitative relation between the errors in the output and the sampling frequency and the errors in representation.

From now on, we will be studying the uniform convergence properties of behaviors and events that are well-typed and closed. These properties are established by a series of theorems. We will omit or only sketch the proof for the theorems. Complete formal proofs can be found in Appendix A.

First, **time** and **input** uniformly converge unconditionally:

Theorem 4.2 (Time). $\vdash \mathbf{time};\ \mathcal{B} \implies \widetilde{\mathrm{at}}[\![\mathbf{time}]\!]_{\mathcal{B}}\ P^t \rightarrowtail t\big|_{t_0}^{\infty}$.

Theorem 4.3 (Input). $\vdash \mathbf{input};\ \mathcal{B} \implies \widetilde{\mathrm{at}}[\![\mathbf{input}]\!]_{\mathcal{B}}\ P^t \rightarrowtail \mathrm{input}(t)\big|_{t_0}^{\infty}$.

If the function being lifted is uniformly continuous, and the sub-behaviors are all uniformly convergent, then the result behavior of **lift** is also uniformly convergent:

Theorem 4.4 (Lift). Given $b \equiv \mathbf{lift}\ (\lambda(x_1, \cdots, x_n).e)\ b_1\ \cdots\ b_n$, if we have:

1. $\vdash b;\ \mathcal{B}$,

2. $\lambda(y_1, \cdots, y_n).\mathbf{eval}[\![e]\!]_{\{x_j \mapsto y_j\}}$ is uniformly continuous, and

3. $\forall j \in \mathbb{N}_n.\ \widetilde{\text{at}}[\![b_j]\!]_{\mathcal{B}}\ P^t \rightarrowtail f_j(t)\big|_{t_0}^{T} - S_j$,

then

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail \mathbf{eval}[\![e]\!]_{\{x_j \mapsto f_j(t)\}}\Big|_{t_0}^{T} - \bigcup_{j=1}^{n} S_j.$$

The above theorem has a particularly simple case for lifting of constants:

Corollary 4.1 (Lift0). $\vdash \mathbf{lift}\ e;\ \mathcal{B} \implies \widetilde{\text{at}}[\![\mathbf{lift}\ e]\!]_{\mathcal{B}}\ P^t \rightarrowtail \mathbf{eval}[\![e]\!]_{\emptyset}\big|_{t_0}^{\infty}$.

Proof. This is just the special case for Theorem 4.4 when $n = 0$. □

To present Theorem 4.5 and 4.8 more concisely, we define the notation

$$t \lessdot t'$$

for that either $t < t'$ or $t = t' = 0$, where $t, t' \in \mathbb{T}$. The intuition is that we want t to be smaller than t' — unless it cannot be any smaller since t' already has the smallest possible value 0, in which case we want t to be 0 too.

We show that **integral** preserves uniform convergency:

Theorem 4.5 (Integral). If

1. $\vdash \mathbf{integral}\ b;\ \mathcal{B}$,

2. $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\big|_{t_0}^{T} - S$ where S is finite, and

3. there is $T_0 \lessdot t_0$ such that

   (a) for all time sequence $P^t = \langle t_1, \cdots, t\rangle$ where $T_0 \le t_1 \le t \le T$, $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t$ is bounded,

   (b) $f(t)$ is bounded on $[T_0, T]$, and

   (c) $\int_{T_0}^{T} f(t)\mathrm{d}t \neq \bot$,

then we have

$$\widetilde{\text{at}}[\![\textbf{integral}\ b]\!]_{\mathcal{B}}\ P^t \rightarrowtail \int_{t_0}^{t} f(\tau)\mathrm{d}\tau\Big|_{t_0}^{T}.$$

The **let-behavior** construct also preserves the uniform convergency of its body:

Theorem 4.6 (Let-behavior). Given $b \equiv$ **let behavior** $\{z_j(x_j) = w_j\}$ **in** b', we have that

$$\vdash b;\ \mathcal{B} \text{ and } \widetilde{\text{at}}[\![b']\!]_{\mathcal{B}\cup\{z_j \mapsto \lambda x_j.w_j\}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T} - S$$

imply that

$$\widetilde{\text{at}}[\![b']\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T} - S.$$

The switcher is more involved. Roughly speaking, if the initial behavior, the triggering events, and the after behavior are all uniformly convergent, then the whole behavior is uniformly convergent:

Theorem 4.7 (Until). Given $b \equiv b'$ **until** $\langle ev_j \Rightarrow z_j \rangle^{j\in\mathbb{N}_n}$, and $\vdash b;\ \mathcal{B}$,

1. if $\forall j \in \mathbb{N}_n.\ \widetilde{\text{occ}}^*[\![ev_j]\!]_{\mathcal{B}}\ t_0\ T = \mathsf{Nothing}$ and $\widetilde{\text{at}}[\![b']\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T} - S$, then

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T} - S;$$

2. if $\exists T' \in (t_0, T],\ k \in \mathbb{N}_n$, such that

   (a) $\widetilde{\text{occ}}^*[\![ev_k]\!]_{\mathcal{B}}\ t_0\ T' = \mathsf{Just}\ (\tau, v)$,

   (b) $\forall j \in \mathbb{N}_n - \{k\}.\ \widetilde{\text{occ}}^*[\![ev_j]\!]_{\mathcal{B}}\ t_0\ T' = \mathsf{Nothing}$,

   (c) $\widetilde{\text{at}}[\![b']\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{\tau} - S$,

   (d) $\widetilde{\text{at}}[\![w[x := u]]\!]_{\mathcal{B}}\ P^t \rightarrowtail g(u,t)\Big|_{\tau}^{T} - S'$, where $\lambda x.w \equiv \mathcal{B}(z_k)$, and

   (e) $\forall \epsilon > 0.\ \exists \delta > 0.\ \forall t \in [\tau, T].\ |\, u - v \,| < \delta \implies |\, g(u,t) - g(v,t) \,| < \epsilon$,

   then

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail h(t)\Big|_{\tau}^{T} - (S \cup S' \cup \{\tau\})$$

where $h(t) = \text{if } t < \tau \text{ then } f(t) \text{ else } g(v, t)$.

If b uniformly converges, so does **when** b:

Theorem 4.8 (When). Given $ev \equiv$ **when** b and $\vdash ev;\ \mathcal{B}$, we have

1. if $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\big|_{t_0}^{T}$, and there are T_0, T_1 where $T_0 \lessdot t_0 \leq T_1 \leq T$, such that

   (a) $T_0 \leq t < T_1 \implies f(t) =$ **True**, and

   (b) $T_1 < t < T \implies f(t) =$ **False**,

   then $\widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T = \mathsf{Nothing}$;

2. if there are $T_0, T_1, T_2, T_3 \in \mathbb{T}$ where $T_0 \lessdot t_0 \leq T_1 < T_2 < T_3 \leq T$, such that

   (a) $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\big|_{t_0}^{T_3}$,

   (b) $t \in [T_0, T_1) \cup (T_2, T_3] \implies f(t) =$ **True**, and

   (c) $t \in (T_1, T_2) \implies f(t) =$ **False**,

   then $\widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T = \mathsf{Just}\ (T_2, ())$.

If the function being lifted is continuous, and the event and sub-behaviors are uniformly convergent, then the result event of **liftev** is uniformly convergent:

Theorem 4.9 (Liftev). Given $ev \equiv$ **liftev** $(\lambda(x_0, \cdots, x_n).e)\ ev'\ b_1\ \cdots\ b_n$ and $\mathcal{B}$ where $\vdash ev;\ \mathcal{B}$, write oc for $\widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T$ and oc' for $\widetilde{\text{occ}}^*[\![ev']\!]_{\mathcal{B}}\ t_0\ T$. We have

1. $oc' = \mathsf{Nothing} \implies oc = \mathsf{Nothing}$, and

2. if

   (a) $oc' = \mathsf{Just}\ (\tau, v_0)$,

   (b) there is $\xi > 0$ such that for all $j \in \mathbb{N}_n$,

      i. $\widetilde{\text{at}}[\![b_j]\!]_{\mathcal{B}}\ P^t \rightarrowtail f_j(t)\big|_{t_0}^{T} - S_j$ where $\tau \notin [S_j]_\xi$, and

      ii. $f_j(t)$ is continuous at $t = \tau$,

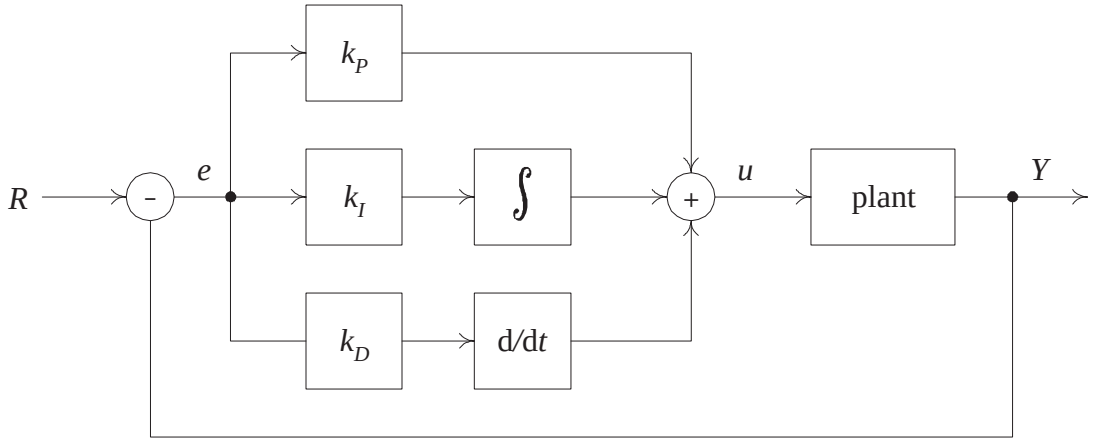
and

(c) the function $g(y_0, y_1, \cdots, y_n) = \textbf{eval}[\![e]\!]_{\{x_j \mapsto y_j\}^{j \in \{0..n\}}}$ is continuous at $(v_0, v_1, \cdots, v_n)$, where $\forall j \in \mathbb{N}_n.\ v_j = f_j(\tau)$,

then $oc = \mathsf{Just}\ (\tau, g(v_0, v_1, \cdots, v_n))$.

4.6 Applications of H-FRP

As an example of using H-FRP, this section discusses how we can expressive a commonly-used form of controllers.



In control systems, one important category of controllers is *PID controllers*, or Proportional-Integral-Derivative controllers. The above diagram shows such a controller, where R is the input to the system, and represents the desired output of the plant; Y is the actual output of the plant; e is the difference between Y and R: the error; the controller adds up a signal proportional to e, a signal proportional to the integral of e, and a third signal proportional to the derivative of e, to obtain the control signal u. By adjusting the three coefficients k_P, k_I, and k_D, the stability and other characteristics of the controller can be tuned.

With a small extension to H-FRP, a PID controller can be expressed as:

$$\begin{array}{l}\textbf{let behavior}\ \mathsf{e} = \textbf{lift}\ (-)\ R\ Y \\ \textbf{in lift}\ (\lambda(\mathsf{p},\mathsf{i},\mathsf{d}).\ k_P \cdot \mathsf{p} + k_I \cdot \mathsf{i} + k_D \cdot \mathsf{d})\ \mathsf{e}\ (\textbf{integral}\ \mathsf{e})\ (\textbf{derivative}\ \mathsf{e})\end{array}$$

where **derivative** is easily defined by a translation to RT-FRP as:

$$\begin{array}{rcl}\textbf{tr}[\![\textbf{derivative}\ b]\!] & = & \textbf{let snapshot}\ x \leftarrow \textbf{tr}[\![b]\!]\ \textbf{in} \\ & & \quad \textbf{let snapshot}\ \mathsf{t} \leftarrow \textbf{time in} \\ & & \quad\quad \textbf{let snapshot}\ (\mathsf{x0},\mathsf{t0}) \leftarrow \\ & & \quad\quad\quad \textbf{delay}\ (0,0)\ (\textbf{ext}\ (x,\mathsf{t})) \\ & & \quad\quad \textbf{in ext}\ (\textbf{if}\ \mathsf{t0} < 0\ \textbf{then}\ 0\ \textbf{else}\ (x - \mathsf{x0})/(\mathsf{t} - \mathsf{t0}))\end{array}$$

where x is a fresh variable.

Chapter summary

H-FRP is a subset of RT-FRP designed for hybrid real-time reactive systems. It has a continuous-time based denotational semantics natural for modeling and reasoning about hybrid systems, and a discrete operational semantics that can be executed on a digital computer. We show that under suitable conditions, the operational semantics of an H-FRP program converges to its denotational semantics. Most programs are expected to satisfy these conditions. Compared with our previous work [53], the conditions we present in this chapter are more accurate and are satisfied by a larger class of programs.

Chapter 5

Event-driven FRP

When a reactive system is programmed in RT-FRP, it is driven by a single stream of inputs. The system advances in steps. At each step, the run-time engine samples (or polls) the environment to get the current input. Since in general there is no way to statically determine which part of the system will be affected by the input, the whole system is traversed to propagate this information, which is used to update the system state and generate the response. Usually the system is run as fast as possible or at some fixed pace deemed fast enough.

Among all reactive systems, a large number are purely driven by multiple event sources: such a system updates its internal state and changes its output only when an external event occurs; between two event occurrences, the system does nothing. We call these systems *event-driven*. One example of event-driven reactive systems is interrupt-driven micro-controllers, which we encounter in building soccer-playing robots for the RoboCup competition [46].

Traditionally, event-driven systems have been programmed using imperative languages in a style known as *event-handling*: for each type of events, the user defines a call-back function or procedure called the *event handler* (or the *listener*); when an event occurs, its handler is invoked to update the system state and the output. Each event handler only needs to change the part of the system state affected by that particular event. When no

event is occurring, the system does nothing.

In practice, this programming paradigm works well but has certain limitations (among them scalability problems) that make it difficult for programming and maintaining large scale systems (we will come back to these limitations later).

By using different input values to denote different events, we can program event-driven systems in the RT-FRP framework. However, this is not always a good idea: in RT-FRP the whole system is traversed at each step to propagate the input, even if only a small part of the system is interested in the current event. For this reason, in applications where performance is critical (such as micro-controllers), RT-FRP is often not suitable.

This chapter addresses this issue by defining a restricted form of RT-FRP called *E-FRP* (for *Event-Driven* FRP), which does not have many of the problems the event-handling approach suffers. In this language, an *event impact analysis* is used to determine which part of the system is affected by which event. We also define a strategy for compiling E-FRP to low-level code, and prove the correctness of the compilation. By incorporating the result of the event impact analysis, the target code is made efficient.

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, and for writing controllers in a robot-simulation computer game named MindRover [45], which is intended by its creator to be programmed in the object-oriented language ICE [39].

In this chapter, we present the E-FRP language and its compilation, with an example from our work on the RoboCup challenge.

5.1 Motivation

Before jumping into the details of the E-FRP language and how we compile it, it is important to understand the nature of the problem we are trying to solve with E-FRP. This is the goal of this section.

Our motivation for developing E-FRP comes from our work on building a team of robots for the RoboCup soccer competition [46]. We found that programming the low-level controllers on-board the robots in C is error-prone and tedious. Since the high-level controllers are programmed in FRP, reasoning about the whole system is also a problem. We tried to solve the problems by designing a language that is resource-bounded, efficient, and natural for programming event-driven systems.

5.1.1 Physical model

First we describe the hardware profile of our robots.

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.

Each robot is equipped with a radio receiver to get commands from the central controller and a PIC16C66 micro-controller [33] to carry out the commands. PIC16C66 is a primitive RISC processor that runs at a slow frequency and has only 4KB of RAM. Hence the program has to be very efficient in CPU cycle and memory usage.

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 on and off (so the motor is in either the on or the off mode at any time). 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.

5.1.2 The Simple RoboCup (SRC) controller

One of the tasks of the low-level controller running on PIC16C66 is to regulate the power being sent to the motor, based on the speed sensor feedback and increase / decrease speed instructions coming from the central FRP controller, so as to maintain a desired speed for the wheel. We call this component the *Simple RoboCup* (*SRC*) controller, and describe it in this section.

Figure 5.1 presents the block diagram of SRC, where we adopt the following convention:

- a dotted line carries an event;
- a solid line carries a behavior;
- a rectangle is a *stateful behavior generator* whose state can be updated by events;
- a shaded rectangle with round corners is a *stateless behavior generator* whose output is a function of its input behaviors; behavior generators sometimes are also called *modules*; and
- a small dark triangular mark at the end of a dotted line means the event affects this stateful module "later." This annotation is useful when one event affects more than one module and we need to specify the order in which they are updated. In this example, both speed and duty cycle react to the event ClkSlow, but when ClkSlow occurs, duty cycle is updated first.

There are five behaviors in this control system:

- desired speed and speed are the desired speed and the measured speed of the wheel, respectively;
- duty cycle is a number between 0 and 100 determining what percentage of time the motor should be on;

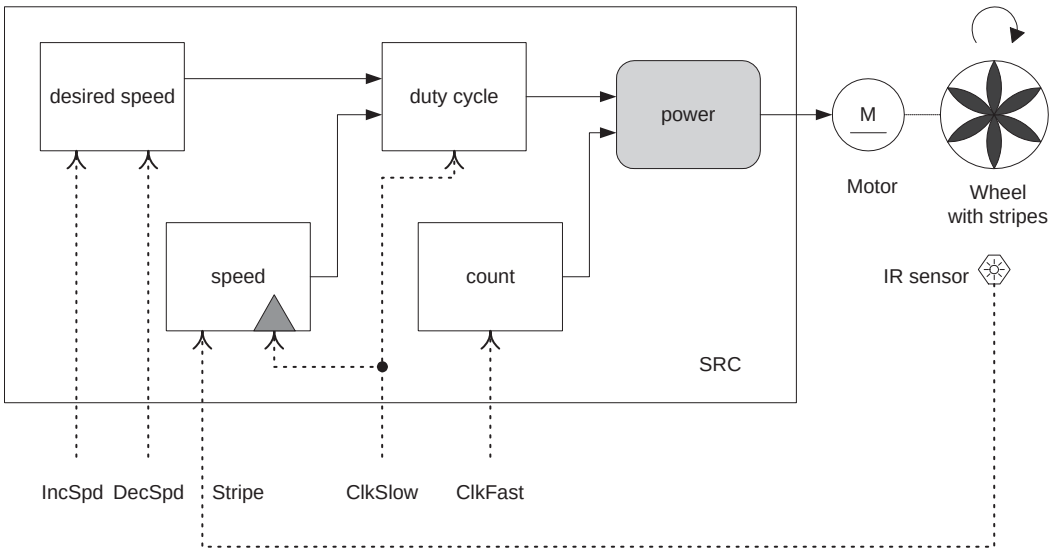


Figure 5.1: Simple RoboCup Controller: block diagram

- count is a counter that repeatedly goes from 0 to 100; and
- power is a Boolean behavior which determines the motor's on / off status.

These behaviors are driven by five independent event sources:

- IncSpd and DecSpd increment and decrement the desired speed;
- Stripe occurs 32 times for every full revolution of the wheel;
- ClkFast and ClkSlow are two clocks that tick at regular, but different, intervals. The frequency of ClkFast is much higher than that of ClkSlow.

When the IR sensor detects a stripe passing under it, it generates a Stripe event, which triggers the speed behavior to increment. With each ClkSlow occurrence, speed is reset to 0. Since ClkSlow occurs at regular intervals, speed counts the number of Stripe's in a fixed period of time, which corresponds to the actual wheel speed. When ClkSlow occurs, before speed is reset, its value is compared with desired speed to determine duty cycle. The triangular "delay" mark makes sure duty cycle is updated before speed. The behavior

count increments each time ClkFast occurs, and resets to 0 after reaching 100. Finally, power is determined by comparing duty cycle and count, and is updated when either of its inputs is updated.

5.1.3 Programming SRC with event handlers

Initially, this low-level controller was programmed in a specialized subset of C, for which a commercial compiler targeting PIC16C66 exists. The essence of the code can be found in Figure 5.2, where `s`, `ds`, and `dc` are used for speed, desired speed, and duty cycle respectively.

```
void init() {
    ds = s = dc = count = power = 0;
}

void on_IncSpd() {
    ds++;
}

void on_DecSpd() {
    ds--;
}

void on_Stripe() {
    s++;
}

void on_ClkSlow() {
    dc = dc < 100 && s < dc ? dc + 1
       : dc > 0   && s > ds ? dc - 1
       : dc;
    power = count < dc;
    s = 0;
}

void on_ClkFast() {
    count = count >= 100 ? 0 : count + 1;
    power = count < dc;
}
```

Figure 5.2: The SRC controller in C

Even this fairly small example reveals some problems with the event-handling approach for programming event-driven systems:

1. The modularity needs improvement.

   If you try to find the definition for the speed module in Figure 5.2, where it is called `s`, you will notice that the definition is scattered among three functions: `init()`, `on_Stripe()`, and `on_ClkSlow()`. This is against the software engineering principle of providing logically indivisible information in one single place.

   In general, to implement a behavior b that is affected by n events, the programmer needs to insert code to all of the n event handlers. Consequently, to understand b, the reader has to walk through the whole program to find every place b is updated, which is tedious and error-prone. Worse yet, if the programmer decides to drop in some new modules to the system (for example, the same controller is duplicated to drive another wheel), he will have to patch existing event handlers with new pieces of code[1]. The program loses its modularity here.

   Control engineers model event-driven systems using block diagrams, and we should therefore make the transition from these diagrams to programs as smooth as possible. The event-handling approach leaves a lot to be desired here.

2. Code is sometimes duplicated.

   We notice that the line "`power = count < dc;`" appears in both `on_ClkSlow()` and `on_ClkFast()`. In general, if the inputs of a stateless behavior b are affected by n events, then the code for updating b will be repeated n times in all those event handlers. When the definition of b needs to be changed, the programmer has to propagate the change to all of the n event handlers. The amount of effort grows

[1] In the Java event model, more than one listener can subscribe to an event. Hence a Java programmer can extend an event listener (or a group of them) without modifying existing code by registering a new listener with the same event. However, this is not satisfactory either, since to understand the system, the reader now needs to study both the definitions of multiple event listeners and the order they are registered. As the definition of a listener and the registration of it are in different parts of the program, the modularity is broken.

with the size of the system. It will not surprise us if some event handlers are missed or other mistakes are made in this process.

In an ideal programming language, the user only needs to specify the relation between the input and the output of a stateless behavior generator *once*, and should be able to change the definition later by making easy local changes, which is particularly beneficial for large-scale systems.

3. The order of assignments can be confusing.

   In each event handler, a series of assignments are executed to update a group of behaviors. While sometimes the order of these assignments is essential, sometimes it is irrelevant. For example, we cannot change the function `on_ClkSlow()` to

```
void on_ClkSlow() {
    power = count < dc;
    dc = dc < 100 && s < dc ? dc + 1
       : dc > 0   && s > ds ? dc - 1
       : dc;
    s = 0;
}
```

   while it is OK to write it as

```
void on_ClkSlow() {
    dc = dc < 100 && s < dc ? dc + 1
       : dc > 0   && s > ds ? dc - 1
       : dc;
    s = 0;
    power = count < dc;
}
```

Although the effect of assignment order is clear for small systems like SRC, it can be rather confusing for a system that involves hundreds of behaviors. In fact, what is important is *what* the behaviors are updated to, not *how*. The burden for specifying the order of assignments is just an artifact of the imperative programming paradigm, and we would like to be able to avoid this over-specification.

4. While C can generate very efficient target code, it is a poor language to write resource-bounded programs in (any experienced C programmer knows how difficult it is to fight those dangling pointers and leaked memory). For a system with fairly limited processing power and hard requirement on response time, the ability to guarantee resource bounds is very important.

5. Finally, it is difficult to reason about the combination of the main FRP controller and these separate controllers in C. This problem would not exist if the controllers are all written in FRP, or a subset thereof.

5.2 The E-FRP language

E-FRP is our attempt to solve the problem of programming event-driven real-time reactive systems naturally, efficiently, and reliably. It 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.

Before presenting the details, some basic features of the E-FRP execution model need to be explained.

5.2.1 Execution model

As with RT-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 RT-FRP behaviors that can change whenever the system is sampled, E-FRP behaviors can change value only when an event occurs.

In RT-FRP, events are just behaviors of Maybe α types, and can be defined in the normal way a behavior is defined. In E-FRP, the user declares a finite set of events that will be used in the program, and cannot create new events at run-time. On different target platforms, events may have different incarnations, like OS messages, or interrupts (as in the case of micro-controllers). E-FRP events are mutually exclusive, meaning that no two events ever occur simultaneously. Such are the same assumptions used by the event-handling approach. These design decisions avoid potentially complex interactions between event handlers, and thus greatly simplifies the semantics and the compiler. For simplicity, E-FRP events do not carry values with their occurrences, though there is no technical difficulty in making them do so.

There are two kinds of behaviors: stateless and stateful. A *stateless behavior* e can be viewed as a pure function whose inputs are other behaviors, or it can be a constant when there is no input. The value of e depends solely on its current input, and is not affected by the history of the input at all. In this sense, e is like a combinatorial circuit. A *stateful behavior sf* has a state that can be updated by event occurrences. The value of *sf* depends on both the current input and its state. Therefore *sf* is analogous to a sequential circuit.

A program defines a collection of behaviors. 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.

$\boxed{\varphi, H, \mathit{sf}, d, P}$

Phases	$\Phi \ni \varphi$	$::=$	$\varepsilon \mid$ **later**
Event handlers	H	$::=$	$\{I_j \Rightarrow e_j\ \varphi_j\}$
Stateful behaviors	sf	$::=$	**init** $x = v$ **in** H
Behaviors	d	$::=$	$e \mid \mathit{sf}$
Programs	P	$::=$	$\{x_j = d_j\}$

$\boxed{FV(\mathit{sf})}$

$$FV(\textbf{init}\ x = v\ \textbf{in}\ \{I_j \Rightarrow e_j\ \varphi_j\}) \equiv \bigcup_j FV(e_j) - \{x\}$$

Figure 5.3: Syntax of E-FRP and definition of free variables

5.2.2 Syntax

Figure 5.3 defines the syntax of E-FRP and the notion of free variables. The syntactic category I is for event names drawn from a finite set $\mathcal{I}$. The category ε stands for the empty term. 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.

The terms e and v are expressions and values of the base language respectively. As we are using BL as the base language in this thesis, they have been defined in Section 2.1. The only thing to note is that e is also called a "stateless behavior" in E-FRP, while it is called an expression in BL.

A stateful behavior $\mathit{sf} \equiv$ **init** $x = v$ **in** $\{I_j \Rightarrow e_j\ \varphi_j\}$ initially has value v, and changes to the *current* value of e_j when event I_j occurs. Note that x is bound to the old value of *sf* and can occur free in e_j. As mentioned earlier, the execution of an E-FRP program happens in two phases. Depending on whether φ_j is ε or **later**, the change of value takes place in either the first or the second phase, respectively.

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

$FV(d)$ denotes the free variables in behavior d. As d can be either e or *sf*, and $FV(e)$ has been defined in Section 2.2.2, we only need to define $FV(\mathit{sf})$ here.

5.2.3 Type system

The type system of E-FRP is simple and established on the basis of the type system of the base language.

Typing judgments

To avoid clutter, we choose to reuse the same typing judgment notation we used for the base language. In other words, while we write $\Gamma \vdash e : \alpha$ for "e is an expression of type α in context Γ," we will write

$$\Gamma \vdash d : \alpha$$

for "d is a behavior of type α in context Γ."

Since the category d is defined as

$$d ::= e \mid sf,$$

the definition of $\Gamma \vdash d : \alpha$ can be presented in two parts, namely $\Gamma \vdash e : \alpha$ and $\Gamma \vdash sf : \alpha$. The definition of the former is the same as in Section 2.2, which justifies the overloading of the notation. Depending on the context, $\Gamma \vdash e : \alpha$ can be read either "e is an expression of type α" or "e is a stateless behavior of type α."

The meaning of $\Gamma \vdash sf : \alpha$ is defined in Figure 5.4. The same figure also defines

$$\Gamma \vdash P$$

which can be read "context Γ assigns types to behaviors in P."

Acyclicity requirement

We must point out that an important requirement for valid E-FRP programs is *not* captured by the type system: namely, we do not allow the behavior definitions to be cyclic.[2]

[2] What it exactly means to be cyclic will become clear in Section 5.4.1.

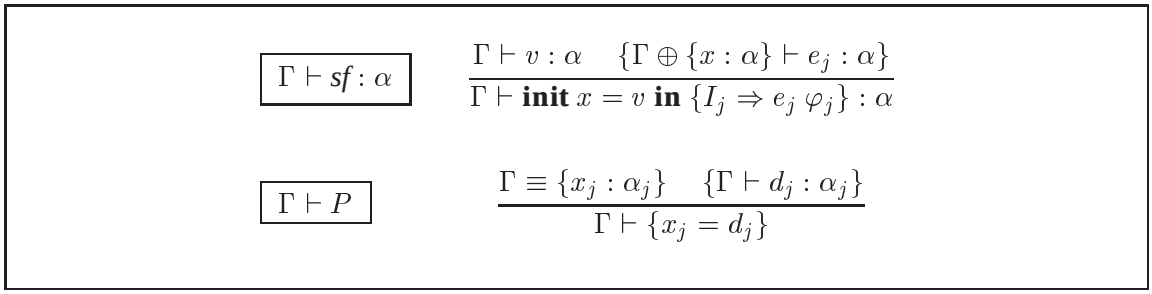


Figure 5.4: Type system of E-FRP

Instead, this check is performed by the compilation process. As we will see in Section 5.4.1, if a program contains cyclic definitions, the compilation will fail. A program is valid only when both it is type-correct and the compilation for it succeeds.

Although we could have encoded the acyclicity requirement using a more advanced type system, the resulting type system would inevitably be complicated and difficult to understand. Besides, it would be redundant since this constraint is already enforced by the compilation process.

5.2.4 Operational semantics

We now give an operational semantics to E-FRP.

Environments

We will continue to use the concept of *environment* defined in Section 2.3. An environment $\mathcal{X}$ maps variables to values, and we repeat its syntax here:

$$\mathcal{X} \quad ::= \quad \{x_j \mapsto v_j\}.$$

Execution judgments

Figure 5.5 defines the operational semantics of an E-FRP program by means of four judgments. The judgments can be read as follows:

- $P \vdash d \xrightarrow{I} v$: "on event I, behavior d yields v;"
- $P \vdash d \xrightarrow{I} d'$: "on event I, behavior d is updated to d';"
- $P \vdash d \xrightarrow{I} v; d'$: "on event I, behavior d yields v, and is updated to d';" This is merely shorthand for $P \vdash d \xrightarrow{I} v$ and $P \vdash d \xrightarrow{I} d'$; and
- $P \xrightarrow{I} \mathcal{X}; P'$: "on event I, program P yields environment $\mathcal{X}$ and is updated to P'."

In the definition of these judgments, we will need to use the judgment $\mathcal{X} \vdash e \hookrightarrow v$ defined in Section 2.3.

$\boxed{P \vdash d \xrightarrow{I} v}$

$$\frac{FV(e) = \{x_j\}^{j \in K} \quad P \equiv P' \uplus \{x_j = d_j\}^{j \in K} \quad \left\{P \vdash d_j \xrightarrow{I} v_j\right\}^{j \in K} \quad \{x_j \mapsto v_j\}^{j \in K} \vdash e \hookrightarrow v}{P \vdash e \xrightarrow{I} v} \text{(e1)}$$

$$\frac{P \vdash e[x := v] \xrightarrow{I} v'}{P \vdash \textbf{init}\ x = v\ \textbf{in}\ \{I \Rightarrow e\} \uplus H \xrightarrow{I} v'} \text{(e2)}$$

$$\frac{H \not\equiv \{I \Rightarrow e\} \uplus H'}{P \vdash \textbf{init}\ x = v\ \textbf{in}\ H \xrightarrow{I} v} \text{(e3)}$$

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

$$\frac{}{P \vdash e \xrightarrow{I} e} \text{(u1)}$$

$$\frac{H \equiv \{I \Rightarrow e\ \varphi\} \uplus H' \quad P \vdash e[x := v] \xrightarrow{I} v'}{P \vdash \textbf{init}\ x = v\ \textbf{in}\ H \xrightarrow{I} \textbf{init}\ x = v'\ \textbf{in}\ H} \text{(u2)}$$

$$\frac{H \not\equiv \{I \Rightarrow e\ \varphi\} \uplus H'}{P \vdash \textbf{init}\ x = v\ \textbf{in}\ H \xrightarrow{I} \textbf{init}\ x = v\ \textbf{in}\ H} \text{(u3)}$$

$\boxed{P \xrightarrow{I} \mathcal{X}; P'}$

$$\frac{\left\{\{x_j = d_j\}^{j \in K} \vdash d_i \xrightarrow{I} v_i; d_i'\right\}^{i \in K}}{\{x_j = d_j\}^{j \in K} \xrightarrow{I} \{x_j \mapsto v_j\}^{j \in K}; \{x_j = d_j'\}^{j \in K}} \text{(p)}$$

Figure 5.5: Operational semantics of E-FRP

When an event I occurs, if a behavior *sf* reacts to I *immediately*, then we update *sf* immediately; if *sf* reacts to I *later*, then we do not update *sf* until all behaviors that depend on *sf* have been updated.

Assuming the base language is deterministic, we can show that the whole E-FRP language is deterministic:

Lemma 5.1 (Deterministic semantics). If the base language is deterministic, then we have that

$$(P \xrightarrow{I} \mathcal{X}; P \text{ and } P \xrightarrow{I} \mathcal{X}'; P') \implies (\mathcal{X} \equiv \mathcal{X}' \text{ and } P \equiv P').$$

The complete proof for this lemma can be found in Appendix A.3. In short, the proof is in two parts, with the first being an induction on the derivation of $P \vdash d \xrightarrow{I} v$, and the second being an induction on the derivation of $P \vdash d \xrightarrow{I} d'$.

5.2.5 SRC in E-FRP

Figure 5.6 presents in E-FRP the SRC controller we introduced in Section 5.1.2. In what follows we explain the definition of each of ds, s, dc, count, and power 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 ClkSlow 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 ClkSlow 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 ClkSlow, the current speed s is compared with the desired speed ds (Recall that s is reset by ClkSlow 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 ClkFast occurs, and is reset every 100 ClkFast interrupts. Hence it counts from 0 to 99 repeatedly. The actual False / True signal going to the motor is determined by the value of power. Since power is True only when count $<$ dc, the larger dc is, the more power is sent to the motor, and hence the more speed. Note that power is updated whenever count or dc changes value.

$$
\begin{aligned}
\mathsf{ds} \quad &= \textbf{init}\ \mathsf{x} = 0\ \textbf{in}\ \{\ \mathsf{IncSpd} \Rightarrow \mathsf{x} + 1, \\
&\qquad\qquad\qquad\quad\ \mathsf{DecSpd} \Rightarrow \mathsf{x} - 1\}, \\
\mathsf{s} \quad &= \textbf{init}\ \mathsf{x} = 0\ \textbf{in}\ \{\ \mathsf{ClkSlow} \Rightarrow 0\ \textbf{later}, \\
&\qquad\qquad\qquad\quad\ \mathsf{Stripe} \Rightarrow \mathsf{x} + 1\}, \\
\mathsf{dc} \quad &= \textbf{init}\ \mathsf{x} = 0\ \textbf{in}\ \{\ \mathsf{ClkSlow} \Rightarrow \textbf{if}\ \mathsf{x} < 100 \wedge \mathsf{s} < \mathsf{ds} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathsf{x} + 1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else if}\ \mathsf{x} > 0 \wedge \mathsf{s} > \mathsf{ds} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathsf{x} - 1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathsf{x}\}, \\
\mathsf{count} \ &= \textbf{init}\ \mathsf{x} = 0\ \textbf{in}\ \{\ \mathsf{ClkFast} \Rightarrow \textbf{if}\ \mathsf{x} \geq 100\ \textbf{then}\ 0 \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathsf{x} + 1\}, \\
\mathsf{power} &= \mathsf{count} < \mathsf{dc}
\end{aligned}
$$

Figure 5.6: The SRC controller in E-FRP

Inspecting this example, we can show that E-FRP solves or makes improvements in each of the problems we mentioned for the event-handling approach in Section 5.1.3:

1. The modularity is improved.

   In the event-handling style, the programmer defines the impact of the events one by one. E-FRP adopts a behavior-oriented view, where the programmer defines behaviors one by one. Our experience is that in practice it is much more likely for a

user to change, add, or remove behaviors than to do the same with events.

However, we understand that there are situations where the event-oriented view is preferred (e.g. when one is interested in the effect on the state induced by a particular event), and we do *not* claim that the behavior-oriented view is always superior (although it is usually the case). Fortunately, our compiler generates the event-oriented view from the behavior-oriented view, so the user can always look at the target code when the event-oriented view is needed.

2. Code duplication for stateless behaviors is eliminated.

   In E-FRP, we only need to define a stateless behavior once, no matter it is affected by how many events.

3. There is no more confusing order of assignments.

   An E-FRP program only specifies the relations between the behaviors and the events. It is not concerned with the exact order in which the behaviors are updated, and relies on the compiler to figure it out. The only exception is when one behavior wants to have access to the value of another behavior before it updates, in which case the programmer inserts a **later** annotation.

   In fact, the order of behavior definitions and event-handler branches in no way affects the semantics of an E-FRP program. Hence the reader is relieved from trying to understand why a particular order is used.

4. The program is guaranteed to be resource-bounded, as we will show later.

5. We are also going to show that E-FRP has an operational semantics in the same style of that of RT-FRP. Thus reasoning about mixed systems programmed in both languages is feasible.

5.3 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.

5.3.1 Syntax

The syntax of SimpleC is defined in Figure 5.7. Again, e is the base language expression.

Assignment sequences	A	$::=$	$\langle x_j := e_j \rangle$
Programs	Q	$::=$	$\{(I_j, A_j, A'_j)\}$

Figure 5.7: Syntax of SimpleC

A program Q is just a collection of event handler function definitions. An event handler has the form (I, A, A'), where I is the event to be handled and also serves as the name of the function. The function body is split into two consecutive parts A and A', which are called *the first phase* and *the second phase* respectively. The idea is that the program Q executes in an environment that defines all free variables in Q, and when an event I occurs, first A is executed to update the environment to match that yielded by the source program, then A' is executed to prepare the environment for the next event.

Note that we reuse the base language expression e here for SimpleC computations. We do this because we are not interested in compiling the base language to a low-level language. In particular, BL is resource-bounded and first-order, and it is not difficult to devise a way for encoding BL expressions and values in C. As we will show later, for our approach of compiling E-FRP to SimpleC to work, we shall require that the values in the base language contain no free variables. In other words, we require that $FV(v) = \emptyset$ for all value v. By Lemma 2.6 in Section 2.4, BL holds this property.

5.3.2 Compatible environments

SimpleC programs are environment transformers: given an initial environment which defines every variable in the program, when an event occurs, the corresponding event handler is executed, which changes the values of the variables in the environment.

Obviously, a SimpleC program Q cannot work with an arbitrary environment $\mathcal{X}$ — the value $\mathcal{X}$ assigns to a variable x must have a type compatible with the usage of x in Q. Hence we define the concepts

- "assignments A is compatible with context Γ" (written $\Gamma \vdash A$),
- "environment $\mathcal{X}$ and assignments A are compatible under context Γ" (written $\Gamma \vdash \mathcal{X}; A$), and
- "environment $\mathcal{X}$ and program Q are compatible under context Γ" (written $\Gamma \vdash \mathcal{X}; Q$)

as

$$\boxed{\Gamma \vdash A} \qquad \frac{\{\Gamma \vdash e_j : \Gamma(x_j)\}}{\Gamma \vdash \langle x_j := e_j \rangle}$$

$$\boxed{\Gamma \vdash \mathcal{X}; A} \qquad \frac{\Gamma \vdash \mathcal{X} \quad \Gamma \vdash A}{\Gamma \vdash \mathcal{X}; A}$$

$$\boxed{\Gamma \vdash \mathcal{X}; Q} \qquad \frac{\Gamma \vdash \mathcal{X} \quad \left\{\Gamma \vdash A_j \quad \Gamma \vdash A'_j\right\}}{\Gamma \vdash \mathcal{X}; \{(I_j, A_j, A'_j)\}}$$

where the judgment $\Gamma \vdash \mathcal{X}$ has been defined in Section 2.3.

5.3.3 Operational semantics

Figure 5.8 gives an operational semantics to SimpleC, where the judgments are read as follows:

- $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$: "executing assignment sequence A updates environment $\mathcal{X}$ to $\mathcal{X}'$."

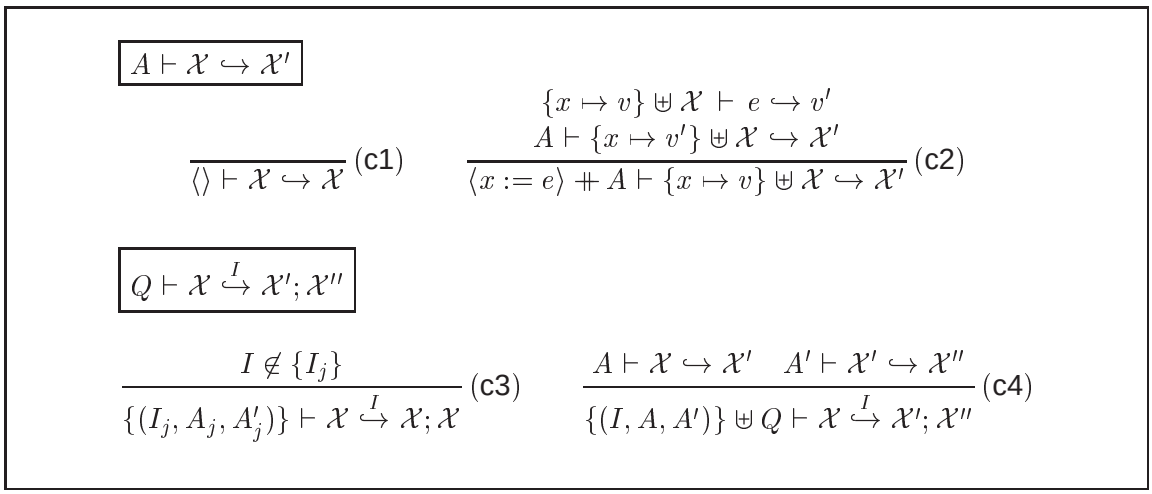


Figure 5.8: Operational semantics of SimpleC

- $Q \vdash \mathcal{X} \overset{I}{\hookrightarrow} \mathcal{X}'; \mathcal{X}''$: "when event I occurs, program Q updates environment $\mathcal{X}$ to $\mathcal{X}'$ in the first phase, then to $\mathcal{X}''$ in the second phase."

The rules are straightforward and do not need explanation. Basically SimpleC has the standard semantics of an imperative language that has no control structure except sequential assignment statements.

5.3.4 Properties

Definition 5.1. Similar to the corresponding concepts for the base language defined in Section 2.4, we say that:

- SimpleC is *productive*, if $\Gamma \vdash \mathcal{X}; A \implies \exists \mathcal{X}'.\ A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$,
- SimpleC is *deterministic*, if $\Gamma \vdash \mathcal{X}; A \wedge A \vdash \mathcal{X} \hookrightarrow \mathcal{X}' \wedge A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'' \implies \mathcal{X}' \equiv \mathcal{X}''$,
- SimpleC is *type-preserving*, if $(\Gamma \vdash \mathcal{X}; A$ and $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}') \implies \Gamma \vdash \mathcal{X}'$, and
- SimpleC is *resource-bounded*, if $(\Gamma \vdash \mathcal{X}; A$ and $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}') \implies$ the size of the derivation tree of $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$ is bounded.

It is easy to show that SimpleC is productive, deterministic, type-preserving, and resource-bounded, given that the base language holds the same properties:

Lemma 5.2 (Productivity). If the base language is productive and type-preserving, then SimpleC is productive.

Proof. By induction on A.

1. $A \equiv \langle\rangle$. By rule c1, we have $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}$ and are done.

2. $A \equiv \langle x := e\rangle +\!\!+ A_1$. Since $\Gamma \vdash \mathcal{X}; A$, we know that $\Gamma \vdash e : \Gamma(x)$. Since $\Gamma \vdash \mathcal{X}$ and the base language is terminating, we know that there are v, v', and $\mathcal{X}_1$ such that $\mathcal{X} \equiv \{x \mapsto v\} \uplus \mathcal{X}_1$ and $\mathcal{X} \vdash e \hookrightarrow v'$. Furthermore, since the base language is type-preserving, we also know that $\Gamma \vdash v' : \Gamma(x)$. Therefore $\Gamma \vdash \{x \mapsto v'\} \uplus \mathcal{X}_1; A_1$. By induction hypothesis, there is $\mathcal{X}'$ such that $A_1 \vdash \{x \mapsto v'\} \uplus \mathcal{X}_1 \hookrightarrow \mathcal{X}'$. According to rule c2, we have that $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$ and are done.

□

Lemma 5.3 (Determinism). If the base language is deterministic, then SimpleC is deterministic.

Proof. Let $D1$ be the derivation tree for $A \vdash \mathcal{X} \overset{\mathcal{X}}{\hookrightarrow}_1$, and $D2$ be that for $A \vdash \mathcal{X} \overset{\mathcal{X}}{\hookrightarrow}_2$. The proof is by induction on $D1$.

1. The last rule used in $D1$ is c1. In this case, $A \equiv \langle\rangle$, and we have $\mathcal{X}_1 \equiv \mathcal{X} \equiv \mathcal{X}_2$.

2. The last rule in $D1$ is c2. There are $x, v, \mathcal{X}_1, e, v'$, and A_1 such that

   (a) $\mathcal{X} \equiv \{x \mapsto v\} \uplus \mathcal{X}_1$,

   (b) $A \equiv \langle x := e\rangle +\!\!+ A'$,

   (c) $\mathcal{X} \vdash e \hookrightarrow v'$, and

   (d) $A_1 \vdash \{x \mapsto v'\} \uplus \mathcal{X}_1 \hookrightarrow \mathcal{X}'$.

   In this case, the last rule in $D2$ must be c2 too, and there must be v'' such that

   (a) $\mathcal{X} \vdash e \hookrightarrow v''$, and

(b) $A_1 \vdash \{x \mapsto v''\} \uplus \mathcal{X}_1 \hookrightarrow \mathcal{X}''$.

However, since the base language is deterministic, we know that $v' \equiv v''$. By induction hypothesis, this leads to that $\mathcal{X}' \equiv \mathcal{X}''$. And we are done.

□

Lemma 5.4 (Type preservation). If the base language is type-preserving, then SimpleC is type-preserving.

Proof. By induction on the derivation of $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$:

1. The last rule used is c1. We have that $A \equiv \langle\rangle$ and $\mathcal{X} \equiv \mathcal{X}'$. It follows trivially that $\Gamma \vdash \mathcal{X}'$.

2. The last rule is c2. We have that there are $\mathcal{X}_1, A_1, x, v, e, v'$ such that

   (a) $\mathcal{X} \equiv \{x \mapsto v\} \uplus \mathcal{X}_1$,

   (b) $A \equiv \langle x := e\rangle +\!\!+ A_1$,

   (c) $\mathcal{X} \vdash e \hookrightarrow v'$, and

   (d) $A_1 \vdash \{x \mapsto v'\} \uplus \mathcal{X}_1 \hookrightarrow \mathcal{X}'$.

   Since $\Gamma \vdash \mathcal{X}; A$, we have that $\Gamma \vdash e : \Gamma(x)$. Since $\Gamma \vdash \mathcal{X}$, $\mathcal{X} \vdash e \hookrightarrow v'$, and the base language is type-preserving, we have $\Gamma \vdash v' : \Gamma(x)$, and therefore $\Gamma \vdash \{x \mapsto v'\} \uplus \mathcal{X}_1$. Hence $\Gamma \vdash \{x \mapsto v'\} \uplus \mathcal{X}_1; A_1$.

   Since $A_1 \vdash \{x \mapsto v'\} \uplus \mathcal{X}_1 \hookrightarrow \mathcal{X}'$, by induction hypothesis, we have that $\Gamma \vdash \mathcal{X}'$.

□

Lemma 5.5 (Resource bound). If the base language is resource-bounded, then SimpleC is resource-bounded.

Proof. Given $\Gamma \vdash \mathcal{X}; A$ and $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$, let n be the length of A, and D be the derivation of $A \vdash \mathcal{X} \hookrightarrow \mathcal{X}'$. We note that in D, we use the rule c1 once and c2 n times. Each time c2

is used, we derive $\mathcal{X}'' \vdash e : v$ for some $\mathcal{X}''$, e, and v. Since we assume the base language is resource-bounded, the size of the derivation of $\mathcal{X}'' \vdash e : v$ is bounded. As there are n such derivations in D, the size of D is also bounded. □

5.4 Compilation from E-FRP to SimpleC

One of our main contributions is a provably correct compiler from E-FRP to SimpleC. This section presents such a compiler implemented in Haskell, and proves that it is correct.

5.4.1 Compilation strategy

A compilation strategy is described by a set of judgments presented in Figure 5.9. The judgments are read as follows:

- $x := e < A$: "e does not depend on x or any variable updated in A."
- $P \vdash_I^1 A$: "A is the first phase of P's event handler for I."
- $P \vdash_I^2 A$: "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 stateful behavior named x. Temporary variables are necessary for compiling certain recursive definitions.

Implicit in this compilation process is an *event impact analysis* that determines which behaviors are affected by a given event I. In the event handler for I, we only need to update those behaviors affected by I.

The compilation relation is clearly decidable, but the reader should 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 will show in Section 5.4.3 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.

5.4.2 Compilation examples

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

$$\begin{aligned} \mathsf{x1} &= \textbf{init}\, \mathsf{x} = 0\ \textbf{in}\ \{I_1 \Rightarrow \mathsf{x} + \mathsf{x2}\}, \\ \mathsf{x2} &= \textbf{init}\, \mathsf{y} = 1\ \textbf{in}\ \{I_2 \Rightarrow \mathsf{y} + \mathsf{x1}\} \end{aligned}$$

Here x1 depends on x2, and x2 depends on x1, 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:

$$\begin{aligned} &(I_1, \quad \langle \mathsf{x1} := \mathsf{x1}^+ + \mathsf{x2} \rangle, \quad \langle \mathsf{x1}^+ := \mathsf{x1} \rangle), \\ &(I_2, \quad \langle \mathsf{x2} := \mathsf{x2}^+ + \mathsf{x1} \rangle, \quad \langle \mathsf{x2}^+ := \mathsf{x2} \rangle) \end{aligned}$$

Consider the following source program:

$$\begin{aligned} \mathsf{x1} &= \textbf{init}\, \mathsf{x} = 0\ \textbf{in}\ \{I \Rightarrow \mathsf{x} + \mathsf{x2}\}, \\ \mathsf{x2} &= \textbf{init}\, \mathsf{y} = 1\ \textbf{in}\ \{I \Rightarrow \mathsf{x1}\ \textbf{later}\} \end{aligned}$$

Here x1 and x2 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

$\boxed{x := e < A}$

$$\frac{FV(e) \cap (\{x\} \uplus \{x_j\}) \equiv \emptyset}{x := e < \langle x_j := e_j \rangle}$$

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

$$\frac{}{\emptyset \vdash_I^1 \langle\rangle}\ (\text{x1}) \qquad \frac{P \vdash_I^1 A \quad x := e < A}{\{x = e\} \uplus P \vdash_I^1 \langle x := e \rangle +\!\!+ A}\ (\text{x2})$$

$$\frac{P \vdash_I^1 A \quad x := e[y := x^+] < A}{\{x = \textbf{init}\, y = v\ \textbf{in}\ \{I \Rightarrow e\} \uplus H\} \uplus P \vdash_I^1 \langle x := e[y := x^+] \rangle +\!\!+ A}\ (\text{x3})$$

$$\frac{P \vdash_I^1 A \quad x^+ := e[y := x] < A}{\{x = \textbf{init}\, y = v\ \textbf{in}\ \{I \Rightarrow e\ \textbf{later}\} \uplus H\} \uplus P \vdash_I^1 \langle x^+ := e[y := x] \rangle +\!\!+ A}\ (\text{x4})$$

$$\frac{P \vdash_I^1 A \quad H \not\equiv \{I \Rightarrow e\ \varphi\} \uplus H'}{\{x = \textbf{init}\, y = v\ \textbf{in}\ H\} \uplus P \vdash_I^1 A}\ (\text{x5})$$

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

$$\frac{}{\emptyset \vdash_I^2 \langle\rangle}\ (\text{x6}) \qquad \frac{P \vdash_I^2 A \quad x := e < A}{\{x = e\} \uplus P \vdash_I^2 \langle x := e \rangle +\!\!+ A}\ (\text{x7})$$

$$\frac{P \vdash_I^2 A}{\{x = \textbf{init}\, y = v\ \textbf{in}\ \{I \Rightarrow e\} \uplus H\} \uplus P \vdash_I^2 \langle x^+ := x \rangle +\!\!+ A}\ (\text{x8})$$

$$\frac{P \vdash_I^2 A}{\{x = \textbf{init}\, y = v\ \textbf{in}\ \{I \Rightarrow e\ \textbf{later}\} \uplus H\} \uplus P \vdash_I^2 \langle x := x^+ \rangle +\!\!+ A}\ (\text{x9})$$

$$\frac{P \vdash_I^2 A \quad H \not\equiv \{I \Rightarrow e\ \varphi\} \uplus H'}{\{x = \textbf{init}\, y = v\ \textbf{in}\ H\} \uplus P \vdash_I^2 A}\ (\text{x10})$$

$\boxed{P \rightsquigarrow Q}$

$$\frac{\{P \vdash_I^1 A_I \quad P \vdash_I^2 A'_I\}^{I \in \mathcal{I}} \qquad \bigcup_j FV(d_j) \subseteq \{x_j\} \text{ where } \{x_j = d_j\} \equiv P}{P \rightsquigarrow \{(I, A_I, A'_I)\}^{I \in \mathcal{I}}}$$

Figure 5.9: Compilation of E-FRP

too, and one acceptable compilation for it is:

$$(I, \langle \mathsf{x1} := \mathsf{x1}^+ + \mathsf{x2};\ \mathsf{x2}^+ := \mathsf{x1} \rangle, \langle \mathsf{x1}^+ := \mathsf{x1};\ \mathsf{x2} := \mathsf{x2}^+ \rangle)$$

However, if the definition for x2 were

$$\mathsf{x2} = \textbf{init}\ \mathsf{y} = 1\ \textbf{in}\ \{I \Rightarrow \mathsf{x1}\}$$

then the code would have been rejected.

If we let both x1 and x2 be updated in the **later** phase, i.e. let the program P be

$$\begin{aligned} \mathsf{x1} &= \textbf{init}\ \mathsf{x} = 0\ \textbf{in}\ \{I \Rightarrow \mathsf{x} + \mathsf{x2}\ \textbf{later}\}, \\ \mathsf{x2} &= \textbf{init}\ \mathsf{y} = 1\ \textbf{in}\ \{I \Rightarrow \mathsf{x1}\ \textbf{later}\} \end{aligned}$$

then one possible translation of P in SimpleC is

$$(I, \langle \mathsf{x1}^+ := \mathsf{x1} + \mathsf{x2};\ \mathsf{x2}^+ := \mathsf{x1} \rangle, \langle \mathsf{x1} := \mathsf{x1}^+;\ \mathsf{x2} := \mathsf{x2}^+ \rangle$$

Finally, we present in Figure 5.10 the target code we got from compiling the SRC controller.

5.4.3 Correctness of compilation

When an event I occurs, a behavior x in an E-FRP program can be updated to have a new value v. 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 v. The following concept is useful for formalizing this intuition:

Definition 5.2 (Program state). The *state* of a program P, written as $\mathrm{state}(P)$, is an envi-

$$
\begin{array}{ll}
(\mathsf{IncSpd}, & \langle \mathsf{ds} := \mathsf{ds}^+ + 1;\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle, \\
& \langle \mathsf{ds}^+ := \mathsf{ds};\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle), \\
(\mathsf{DecSpd}, & \langle \mathsf{ds} := \mathsf{ds}^+ - 1;\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle, \\
& \langle \mathsf{ds}^+ := \mathsf{ds};\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle), \\
(\mathsf{Stripe}, & \langle \mathsf{s} := \mathsf{s}^+ + 1;\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle, \\
& \langle \mathsf{s}^+ := \mathsf{s};\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle), \\
(\mathsf{ClkSlow}, & \langle \mathsf{s}^+ := 0;\ \mathsf{dc} := \textbf{if } \mathsf{dc}^+ < 100 \wedge \mathsf{s} < \mathsf{ds} \textbf{ then } \mathsf{dc}^+ + 1 \\
& \qquad\qquad \textbf{else if } \mathsf{dc}^+ > 0 \wedge \mathsf{s} > \mathsf{ds} \textbf{ then } \mathsf{dc}^+ - 1 \\
& \qquad\qquad \textbf{else } \mathsf{dc}^+; \\
& \ \ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle, \\
& \langle \mathsf{s} := \mathsf{s}^+;\ \mathsf{dc}^+ := \mathsf{dc};\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle), \\
(\mathsf{ClkFast}, & \langle \mathsf{count} := \textbf{if } \mathsf{count}^+ \geq 100 \textbf{ then } 0 \textbf{ else } \mathsf{count}^+ + 1; \\
& \ \ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle, \\
& \langle \mathsf{count}^+ := \mathsf{count};\ \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle)
\end{array}
$$

Figure 5.10: The SRC controller in SimpleC

ronment defined by:

$$\mathrm{state}(P) \equiv \{x_j \mapsto \mathrm{state}_P(e_j)\}^{j\in J} \uplus \{y_j \mapsto \mathrm{state}_P(\mathit{sf}_j), y_j^+ \mapsto \mathrm{state}_P(\mathit{sf}_j)\}^{j\in K}$$

where

- $P \equiv \{x_j = e_j\}^{j\in J} \uplus \{y_j = \mathit{sf}_j\}^{j\in K}$,
- $\mathrm{state}_P(d) = v$ where $P \vdash d \overset{\mathsf{Query}}{\rightharpoonup} v$, and
- $\mathsf{Query} \notin \mathcal{I}$ is a special event whose sole purpose is to reveal the state of the program without updating it; the user cannot use Query in the program.

It is obvious that $\mathrm{state}(-)$ possesses the following properties:

Lemma 5.6. If $\Gamma \vdash P$ and $P \rightsquigarrow Q$, then $\mathrm{state}(P)$ is uniquely defined.

Lemma 5.7. Given $P \equiv \{x_j = e_j\}^{j\in J} \uplus \{y_j = \mathit{sf}_j\}^{j\in K}$, if $\Gamma \vdash P$ and $P \rightsquigarrow Q$, then $\Gamma \uplus \left\{y_j^+ : \Gamma(y_j)\right\}^{j\in K} \vdash \mathrm{state}(P); Q$.

The following lemma shows that if a program P can be compiled, then it can be executed at the source level. This means the compiler does not accept invalid code.

Lemma 5.8. If $\Gamma \vdash P$ and $P \rightsquigarrow Q$, then there are $\mathcal{X}$ and P' such that $P \xrightarrow{I} \mathcal{X}; P'$.

Furthermore, if P compiles to Q, then Q is a valid program that can be executed successfully under environment $\mathrm{state}(P)$:

Lemma 5.9. If $\Gamma \vdash P$ and $P \rightsquigarrow Q$, then there are $\mathcal{X}_1$ and $\mathcal{X}_2$ such that $Q \vdash \mathrm{state}(P) \overset{I}{\hookrightarrow} \mathcal{X}_1; \mathcal{X}_2$.

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 5.1 (Source-target correspondence). Given that $\Gamma \vdash P$, $P \rightsquigarrow Q$, and $P \xrightarrow{I} \mathcal{X}; P'$, we have

1. $P' \rightsquigarrow Q$; and
2. if $Q \vdash \mathrm{state}(P) \overset{I}{\hookrightarrow} \mathcal{X}'; \mathcal{X}''$, then $\mathcal{X}' \supseteq \mathcal{X}$ and $\mathcal{X}'' \equiv \mathrm{state}(P')$.

The second part of the theorem can be illustrated by the figure below:

$$\begin{array}{ccc} P & \xrightarrow{I} & \mathcal{X};\ P' \\ \rightsquigarrow\downarrow & \downarrow\mathrm{state} & \subseteq\downarrow \quad \downarrow\mathrm{state} \\ Q \vdash \mathcal{X}_0 & \overset{I}{\hookrightarrow} & \mathcal{X}';\ \mathcal{X}'' \end{array}$$

The proof of this theorem is rather lengthy and therefore omitted here. Interested readers can find it in Section A.3 of the Appendix. The core of the proof is an induction on the derivation of the compilation process.

Writing

$$P \xrightarrow{I_1} \mathcal{X}_1; P_1 \xrightarrow{I_2} \cdots \xrightarrow{I_n} \mathcal{X}_n; P_n$$

as shorthand for $P \xrightarrow{I_1} \mathcal{X}_1; P_1$, $P_1 \xrightarrow{I_2} \mathcal{X}_2; P_2$, ..., and $P_{n-1} \xrightarrow{I_n} \mathcal{X}_n; P_n$, and

$$Q \vdash \mathcal{X}_0 \xrightarrow{I_1} \mathcal{X}'_1; \mathcal{X}''_1 \xrightarrow{I_2} \cdots \xrightarrow{I_n} \mathcal{X}'_n; \mathcal{X}''_n$$

for $Q \vdash \mathcal{X}_0 \xrightarrow{I_1} \mathcal{X}'_1; \mathcal{X}''_1$, $Q \vdash \mathcal{X}_1 \xrightarrow{I_2} \mathcal{X}'_2; \mathcal{X}''_2$, and $Q \vdash \mathcal{X}_{n-1} \xrightarrow{I_n} \mathcal{X}'_n; \mathcal{X}''_n$, we show that for any input event sequence, the result of the object code step-wise matches that of the source code:

Corollary 5.1 (Compilation correctness). Given that $\Gamma \vdash P$, $P \rightsquigarrow Q$, $P \xrightarrow{I_1} \mathcal{X}_1; P_1 \xrightarrow{I_2} \cdots \xrightarrow{I_n} \mathcal{X}_n; P_n$, and $Q \vdash \mathcal{X}_0 \xrightarrow{I_1} \mathcal{X}'_1; \mathcal{X}''_1 \xrightarrow{I_2} \cdots \xrightarrow{I_n} \mathcal{X}'_n; \mathcal{X}''_n$, we have that for all $j \in \mathbb{N}_n$, $\mathcal{X}'_j \supseteq \mathcal{X}_j$ and $\mathcal{X}''_j \equiv \text{state}(P_j)$.

The corollary can be illustrated by the figure below:

$$\begin{array}{ccccccccc}
P & & \xrightarrow{I_1} & \mathcal{X}_1; P_1 & \xrightarrow{I_2} & \cdots & \xrightarrow{I_n} & \mathcal{X}_n; P_n \\
\rightsquigarrow \downarrow & \downarrow \text{state} & & \subseteq \downarrow & \downarrow \text{state} & & & \subseteq \downarrow & \downarrow \text{state} \\
Q \vdash \mathcal{X}_0 & & \xrightarrow{I_1} & \mathcal{X}'_1; \mathcal{X}''_1 & \xrightarrow{I_2} & \cdots & \xrightarrow{I_n} & \mathcal{X}'_n; \mathcal{X}''_n
\end{array}$$

Proof. The proof is by repeatedly applying part 2 of Theorem 5.1. □

5.4.4 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 of the compiler. In particular, we implemented a compiler that does:

1. Ineffective code elimination: Given a definition $x = d$ in the source program, x is affected by a particular event I only when one of the following is true:

   (a) d is a stateful behavior that reacts to I; or

$\boxed{P \vdash Q \Rightarrow_1 Q'}$

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

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

$\boxed{P \vdash Q \Rightarrow_2 Q'}$

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

$\boxed{P \vdash Q \Rightarrow_3 Q'}$

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

$\boxed{P \vdash Q \Rightarrow_4 Q'}$

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

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

$\boxed{P \vdash Q \Rightarrow_j^* Q'}$

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

$\boxed{P \vdash Q \Rightarrow|_j Q'}$

$$\frac{P \vdash Q \Rightarrow_j^* Q' \quad \nexists Q''.P \vdash Q' \Rightarrow_j Q''}{P \vdash Q \Rightarrow|_j Q'}$$

$\boxed{P \vdash Q \Rightarrow| Q'}$

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

Figure 5.11: Optimization of the target code

(b) d is a stateless behavior, and one of the variables in $FV(d)$ 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

$$\begin{array}{lrcl}
\boxed{RV(A)} & RV(\langle x_j := e_j \rangle) & \equiv & \bigcup_j FV(e_j) \\
\boxed{RV(Q)} & RV(\{(I_j, A_j, A'_j)\}) & \equiv & \bigcup_j \Big(RV(A_j) \cup RV(A'_j) \Big) \\
\boxed{LV(A)} & LV(\langle x_j := e_j \rangle) & \equiv & \{x_j\}
\end{array}$$

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

As an example, Figure 5.12 presents the intermediate and final results of optimizing the SRC program. Stage 1 optimization generates Q_1, where all unnecessary updates to power have been eliminated. Then stage 2 gets rid of all right-hand-side temporary variables in phase 1, as in the change from $\mathsf{ds} := \mathsf{ds}^+ + 1$ to $\mathsf{ds} := \mathsf{ds} + 1$, 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.

As Q_4 is about the same size of the source code, one might be tempted to program in SimpleC directly. However, it is not sensible to make size the only gauge when comparing programs written in different languages. As we have argued, E-FRP abstracts away unnecessary details on evaluation order, offers a declarative view on behaviors, and sometimes eliminates code duplication. Therefore E-FRP is the preferred to SimpleC here.

$$
\begin{array}{ll}
Q_1 \equiv (\mathsf{IncSpd}, & \langle \mathsf{ds} := \mathsf{ds}^+ + 1\rangle, \langle \mathsf{ds}^+ := \mathsf{ds}\rangle),\\
(\mathsf{DecSpd}, & \langle \mathsf{ds} := \mathsf{ds}^+ - 1\rangle, \langle \mathsf{ds}^+ := \mathsf{ds}\rangle),\\
(\mathsf{Stripe}, & \langle \mathsf{s} := \mathsf{s}^+ + 1\rangle, \langle \mathsf{s}^+ := \mathsf{s}\rangle),\\
(\mathsf{ClkSlow}, & \langle \mathsf{s}^+ := 0; \mathsf{dc} := \textbf{if } \mathsf{dc}^+ < 100 \wedge \mathsf{s} < \mathsf{ds} \textbf{ then } \mathsf{dc}^+ + 1\\
& \qquad\qquad \textbf{else if } \mathsf{dc}^+ > 0 \wedge \mathsf{s} > \mathsf{ds} \textbf{ then } \mathsf{dc}^+ - 1 \textbf{ else } \mathsf{dc}^+;\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle,\\
& \langle \mathsf{s} := \mathsf{s}^+;\ \mathsf{dc}^+ := \mathsf{dc}\rangle),\\
(\mathsf{ClkFast}, & \langle \mathsf{count} := \textbf{if } \mathsf{count}^+ \geq 100 \textbf{ then } 0 \textbf{ else } \mathsf{count}^+ + 1;\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle,\\
& \langle \mathsf{count}^+ := \mathsf{count}\rangle)
\end{array}
$$

$$
\begin{array}{ll}
Q_2 \equiv (\mathsf{IncSpd}, & \langle \mathsf{ds} := \mathsf{ds} + 1\rangle, \langle \mathsf{ds}^+ := \mathsf{ds}\rangle),\\
(\mathsf{DecSpd}, & \langle \mathsf{ds} := \mathsf{ds} - 1\rangle, \langle \mathsf{ds}^+ := \mathsf{ds}\rangle),\\
(\mathsf{Stripe}, & \langle \mathsf{s} := \mathsf{s} + 1\rangle, \langle \mathsf{s}^+ := \mathsf{s}\rangle),\\
(\mathsf{ClkSlow}, & \langle \mathsf{s}^+ := 0; \mathsf{dc} := \textbf{if } \mathsf{dc} < 100 \wedge \mathsf{s} < \mathsf{ds} \textbf{ then } \mathsf{dc} + 1\\
& \qquad\qquad \textbf{else if } \mathsf{dc} > 0 \wedge \mathsf{s} > \mathsf{ds} \textbf{ then } \mathsf{dc} - 1 \textbf{ else } \mathsf{dc};\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle,\\
& \langle \mathsf{s} := \mathsf{s}^+;\ \mathsf{dc}^+ := \mathsf{dc}\rangle),\\
(\mathsf{ClkFast}, & \langle \mathsf{count} := \textbf{if } \mathsf{count} \geq 100 \textbf{ then } 0 \textbf{ else } \mathsf{count} + 1;\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle,\\
& \langle \mathsf{count}^+ := \mathsf{count}\rangle)
\end{array}
$$

$$
\begin{array}{ll}
Q_3 \equiv (\mathsf{IncSpd}, & \langle \mathsf{ds} := \mathsf{ds} + 1\rangle, \langle \mathsf{ds}^+ := \mathsf{ds}\rangle),\\
(\mathsf{DecSpd}, & \langle \mathsf{ds} := \mathsf{ds} - 1\rangle, \langle \mathsf{ds}^+ := \mathsf{ds}\rangle),\\
(\mathsf{Stripe}, & \langle \mathsf{s} := \mathsf{s} + 1\rangle, \langle \mathsf{s}^+ := \mathsf{s}\rangle),\\
(\mathsf{ClkSlow}, & \langle \mathsf{dc} := \textbf{if } \mathsf{dc} < 100 \wedge \mathsf{s} < \mathsf{ds} \textbf{ then } \mathsf{dc} + 1\\
& \qquad\qquad \textbf{else if } \mathsf{dc} > 0 \wedge \mathsf{s} > \mathsf{ds} \textbf{ then } \mathsf{dc} - 1 \textbf{ else } \mathsf{dc};\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}; \mathsf{s} := 0\rangle,\\
& \langle \mathsf{s}^+ := \mathsf{s};\ \mathsf{dc}^+ := \mathsf{dc}\rangle),\\
(\mathsf{ClkFast}, & \langle \mathsf{count} := \textbf{if } \mathsf{count} \geq 100 \textbf{ then } 0 \textbf{ else } \mathsf{count} + 1;\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle,\\
& \langle \mathsf{count}^+ := \mathsf{count}\rangle)
\end{array}
$$

$$
\begin{array}{ll}
Q_4 \equiv (\mathsf{IncSpd}, & \langle \mathsf{ds} := \mathsf{ds} + 1\rangle, \langle\rangle),\\
(\mathsf{DecSpd}, & \langle \mathsf{ds} := \mathsf{ds} - 1\rangle, \langle\rangle),\\
(\mathsf{Stripe}, & \langle \mathsf{s} := \mathsf{s} + 1\rangle, \langle\rangle),\\
(\mathsf{ClkSlow}, & \langle \mathsf{dc} := \textbf{if } \mathsf{dc} < 100 \wedge \mathsf{s} < \mathsf{ds} \textbf{ then } \mathsf{dc} + 1\\
& \qquad\qquad \textbf{else if } \mathsf{dc} > 0 \wedge \mathsf{s} > \mathsf{ds} \textbf{ then } \mathsf{dc} - 1 \textbf{ else } \mathsf{dc};\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}; \mathsf{s} := 0\rangle, \langle\rangle),\\
(\mathsf{ClkFast}, & \langle \mathsf{count} := \textbf{if } \mathsf{count} \geq 100 \textbf{ then } 0 \textbf{ else } \mathsf{count} + 1;\\
& \quad \mathsf{power} := \mathsf{count} < \mathsf{dc}\rangle, \langle\rangle)
\end{array}
$$

Figure 5.12: Optimization of the SRC controller

5.5 Translation from E-FRP to RT-FRP

Note that in $P \xrightarrow{I} \mathcal{X}; P'$, P and P' have the same "structure." Their only difference is in the value of v's in the **init-in** construct. Those v's form the internal state of the program, and the output of the program ($\mathcal{X}$) is determined by the input I and the internal state. Therefore a program P in E-FRP can be translated to an equivalent one in RT-FRP as a group of mutually-recursive signals with a internal state.

In general, the program

$$P \equiv \{x_j = e_j\}^{j \in \mathbb{N}_m} \uplus \{y_j = \textbf{init } y'_j = v_j \textbf{ in } H_j\}^{j \in \mathbb{N}_n}$$

translates to

$$\begin{array}{l} \textbf{let snapshot } \mathsf{i} \leftarrow \textbf{input in} \\ \quad \textbf{let snapshot } (y_1, y_1^+, \cdots, y_n, y_n^+) \leftarrow \\ \quad\quad \textbf{delay } (v_1, v_1, \cdots, v_n, v_n)\ (\textbf{ext } e) \\ \quad \textbf{in ext } (e_1, \cdots, e_m, y_1, \cdots, y_n) \end{array}$$

where e is a function on $(y_1, y_1^+, \cdots, y_n, y_n^+)$ which specifies the next state of the program. The current input i will be tested in e to determine which event is occurring. When possible, some or all of the temporary variables $y_1^+, \cdots, y_n^+$ can be optimized away.

We will not describe how to derive e from the source program P, which is possible but tedious. Instead, we translate the SRC controller to RT-FRP as an example:

```
let snapshot i ← input in
  let snapshot (ds, s, dc, count) ←
    delay (0, 0, 0, 0)
      (ext (if i = IncSpd then
              (ds + 1, s, dc, count)
            else if i = Stripe then
              (ds, s + 1, dc, count)
            else if i = DecSpd then
              (ds − 1, s, dc, count)
            else if i = ClkSlow then
              (ds, 0, if dc < 100 ∧ s < ds then dc + 1
                      else if dc > 0 ∧ s > ds then dc − 1
                      else dc, count)
            else if i = ClkFast then
              (ds, s, dc, if count ≥ 100 then 0 else count + 1)
            else
              (ds, s, dc, count)))
  in ext (ds, s, dc, count, if count < dc then 1 else 0)
```

Unlike the translation from H-FRP to RT-FRP, which is an induction on the structure of the program, this translation is not compositional since it involves topological sort and therefore needs global information.

The existence of this translation shows that we can view E-FRP as roughly a subset of RT-FRP.

5.6 Application in games

We have implemented a compiler for E-FRP in Haskell, as well as two back-ends for the compiler. The first back-end translates SimpleC to the C language the PIC16C66 C compiler accepts. Our second back-end generates code in an object-oriented language named ICE [39], which is used to program robots in a computer simulation game called MindRover [45].

As we have been using the Simple RoboCup controller as an example throughout this chapter, the user should already have a good idea on how E-FRP works with microcontrollers. Therefore, we devote this entire section to our interesting experiment on using E-FRP in the MindRover game.

5.6.1 An introduction to MindRover

MindRover is a computer game developed by a company called CogniToy. In the game, the player designs and programs vehicles to accomplish certain tasks, ranging from sports, races, and maze solving to battles against other vehicles provided with the game or created by other players. Once created, the vehicles are put into a simulated world to run on their own. The game renders the scenario in 3-dimensional animation. Whether you can beat the game depends on the vehicle you design and the controller you program for it.

To build a vehicle, you first pick a chassis. There are three kinds to choose from: hovercraft (driven by thrusters), wheeled (car-like), and treaded (tank-like); and there are three sizes of each of them. The large chassis carries more components, but is also heavier and requires more power to move. Obviously in a race heavier is bad. Hence the chassis being chosen is vital for accomplishing the given scenario.

Next you add *physical components* to the chassis for the job at hand. There are two major categories of them: *sensors* and *actuators*. The former category includes radar, sonar, proximity sensor, bumper, radio receiver, and etc. The latter can be further divided into movement (engine, thruster, steering wheel), communication (radio emitter, speaker) and

weapons (laser, machine gun, rocket launcher). You can decide the angle and location for each component. A component has a certain point cost and weight associated with it, so you cannot install unlimited number of them.

Finally, you need to program the vehicle. The details on this come in the next section. For now, the reader just needs to know that MindRover has an event-driven model and the control program of a vehicle amounts to handling various events picked up by the sensors.

5.6.2 Programming vehicles in MindRover

Execution model

The simulated world in MindRover is object-oriented: a component is represented by an object with properties accessible to the user program and the system. For example, a radar object has an angle property that determines the direction of the radar, a scanWidth property that specifies how wide the "fan" of projection is, and a maxRange property for the range it monitors. All these three properties can be read and set by the user program, but there are properties that can only be read, for example sensor readings.

In MindRover, a vehicle interacts with its environment, or in general a component interacts with other components, via event handlers. When an event occurs in the system (for example, the radar sees something, the timer ticks, or a variable is changed), its handler is executed, which may update some properties or trigger more events. When no event occurs, the user program gets no control. Hence this is a event-driven system.

The game supports two ways for programming a vehicle: using a visual building tool or writing the program in a textual language called ICE. Next we briefly introduce both of them.

The visual tool

The easier way of the two is to use the visual tool provided by the game. Suppose we are designing a simple controller that does the following: when some component A generates an event E, the foo property of component B is set to some value v. We can do this in the visual tool by drawing a *wire* (a directed edge) from A to B, right-clicking on the wire and working with a drop-down box to set the **triggeredBy** property of the wire to E, the **updateProperty** property to foo, and the **toValue** property to v.

A wire can have only one source and one destination, and can carry only one action. If we want to do more, more wires are required. For example, if we want to also set the bar property of component C to v' on the same event, we repeat the above process, starting with a wire from A to C.

The visual tool does not allow the user to write expressions directly. If some computation is involved to decide the value of a property, the user has to add *logic components* (Boolean operators, arithmetic operators, variables, timers, and etc) to the system and connect them to other components with wires.

Although the visual language is intuitive and easy to get started with, it is recommended only for casual programming of simple controllers, for the following reasons:

1. Some functionalities are not easily accessible from the visual tool. For example, Boolean operations, arithmetic, and variables all require adding new logic components to the diagram, which soon gets clumsy.

2. Some functionalities of the system are not available at all in the visual tool. For example some system objects can only be used in the ICE language.

3. For a complex controller, too many wires will present in the diagram, obscuring the structure of the system.

4. The diagram does not tell the whole story. A large part of the meaning of a program is hidden among the various properties of the wires and components, making it a

major effort to understand the system.

The ICE programming language

For serious programming, the other method is preferred, which is to write code in the ICE language.

ICE is a simple object-oriented programming language that borrows its syntax from Visual Basic. A vehicle controller in ICE mainly consists of an initializer and a bunch of subroutines as event handlers. The user can also define auxiliary classes and functions. In the initializer, the component objects are created and initialized (for physical components, their positions on the chassis is also set), and the event handlers are registered with their corresponding event sources.

While programming in ICE unleashes the full power of the system, it unfortunately has some big problems too:

1. The ICE program is written in the event-handling approach, thus having all the problems we have discussed in Section 5.1.3. In particular, ICE does not guarantee a resource bound of the program. As an extreme example, an ICE program that has a black hole will even cause the Windows operating system to hang.

2. ICE does not have a well-specified semantics. For example, many key aspects of its execution mechanism is not documented and the ambiguity in the semantics is often resolved in an ad hoc fashion. This makes understanding the program more difficult and lowers the programmer's confidence in a program.

5.6.3 From E-FRP to ICE

We notice that MindRover is an event-driven system, which is the domain of E-FRP. By writing a back-end that translates SimpleC to ICE, we are able to use E-FRP to program MindRover, and avoid the problems with programming in ICE.

We should point out that our work on combining E-FRP and MindRover is not just for fun. Thanks to its simulation of physics, 3D-animation, and a vast selection of components, MindRover provides an economic platform for us to build robotic vehicles and test control algorithms on them. It also serves as a test bed for the applicability of our E-FRP language.

Mapping between ICE and E-FRP

In ICE, events are generated by components, and the syntax $A.I$ is used for the event I generated by component A. By allowing event names to contain ".", ICE events map directly to E-FRP events.

There are two kinds of properties in ICE: read-only and read-write. Read-only properties cannot be set by the user program and hence need no definitions; therefore they are treated as external behaviors in E-FRP. Read-write properties can be updated by the program, and are represented by a normal behavior in E-FRP.

Some behaviors (for example, temporary variables) in the E-FRP program do not correspond to any component properties in ICE. For each such behavior, we allocate an ordinary variable in ICE.

ICE has different syntaxes for assigning to a property and a non-property variable. The former is done by

$$\texttt{call}\ object.\texttt{set}Property(newValue)$$

while the latter is simply

$$variable = newValue.$$

Accordingly, the translation of a SimpleC assignment $x := e$ to ICE depends on whether x is an object property or not.

Extension to E-FRP for firing events

In E-FRP, the generation of events is outside of the model, and cannot be done within the user program. Yet in MindRover, the ability to trigger events in the program is vital. For example, in a battle scenario, the program need to fire a weapon, which is done by triggering the weapon component's Fire event.

We solve this problem by adding a small extension to E-FRP. Namely, we allow a program to contain definitions of the following form:

$$\{I_j\} \textbf{ trigger } I \textbf{ if } e$$

where e is a stateless behavior of Boolean type. Such a definition means that event I is triggered when any event in $\{I_j\}$ occurs *and* the behavior e is True. When $e \equiv$ True, the definition can be written more concisely as

$$\{I_j\} \textbf{ trigger } I.$$

5.6.4 The KillerHover example

We end this section with an example in which a hovercraft is built to track down the opponent and then kill it by launching rockets. We will use E-FRP to program the hovercraft, but first we need to put all necessary physical components on the chassis. We are using three radars, two thrusters, and one rocket launcher. The placement of these component on the chassis is shown in Figure 5.13.

The E-FRP program controlling this vehicle is given by Figure 5.14.

The first three lines set up the front radar. By making the angle property 0, we point the radar to the front direction. In our example, the properties of the radar are all constant numbers. In general, they can be behaviors that change over time, so it is possible to have a rotating radar if that is desired.

The next six lines set up the left and the right radar.

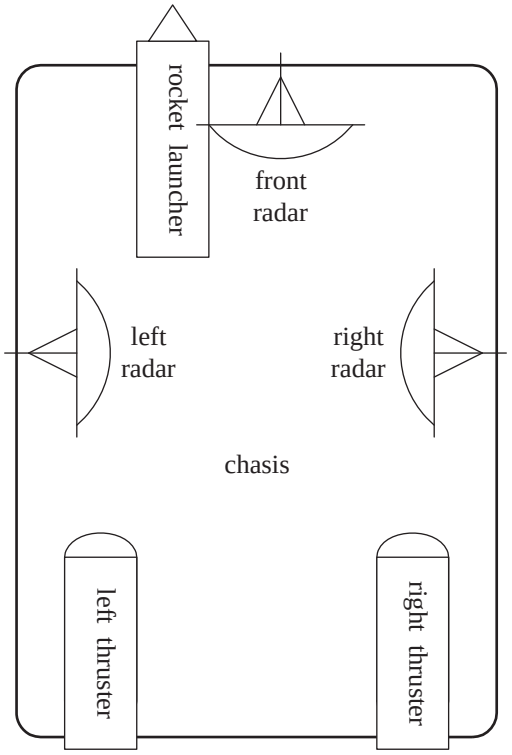


Figure 5.13: Structure of the Killer Hovercraft

$$
\begin{aligned}
&\mathsf{fRadar.angle} &&= 0\\
&\mathsf{fRadar.scanWidth} &&= 20\\
&\mathsf{fRadar.maxRange} &&= 10\\
&\\
&\mathsf{lRadar.angle} &&= 90\\
&\mathsf{lRadar.scanWidth} &&= 20\\
&\mathsf{lRadar.maxRange} &&= 10\\
&\\
&\mathsf{rRadar.angle} &&= -90\\
&\mathsf{rRadar.scanWidth} &&= 20\\
&\mathsf{rRadar.maxRange} &&= 10\\
&\\
&\mathsf{dir} = \textbf{init}\ \_ = \mathsf{Unknown}\ \textbf{in}\ \{\ \mathsf{fRadar.TurnOn} \Rightarrow \mathsf{Front},\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{lRadar.TurnOn} \Rightarrow \mathsf{Left},\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{rRadar.TurnOn} \Rightarrow \mathsf{Right}\}\\
&\\
&\mathsf{lThruster.thrust} = \textbf{if}\ (\mathsf{dir} = \mathsf{Left})\ \textbf{then}\ -50\\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ 50\\
&\\
&\mathsf{rThruster.thrust} = \textbf{if}\ (\mathsf{dir} = \mathsf{Unknown} \vee \mathsf{dir} = \mathsf{Right})\ \textbf{then}\ -50\\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ 50\\
&\\
&\mathsf{fRadar.TurnOn}\ \textbf{trigger}\ \mathsf{rocket.Fire}
\end{aligned}
$$

Figure 5.14: E-FRP program for the Killer Hovercraft

dir is a non-property behavior that represents the current direction of the opponent. It is updated when any of the three radars detects the opponent and thus generates a TurnOn event.

The thrust property of a thruster component determines the amount of power sent to it. The left thruster (lThruster) and the right thruster (rThruster) are sent different amount of power based on the current value of dir, making the vehicle to head for the opponent.

The last line says that the rocket should Fire when the front radar detects the opponent.

5.7 Discussion on later

In E-FRP, the user can specify the order in which behaviors are updated when an event occurs, by annotating certain reactions with the **later** keyword. Yet the reader might have noticed that RT-FRP does not have this feature. Instead, it has a **delay** construct which might be used to achieve similar effect. A natural question to ask is why we invented **later** when designing E-FRP, rather than just copying **delay** from RT-FRP. This section discusses the ramifications of the design options we had and why we picked the current one.

5.7.1 delay of RT-FRP

We could have used the **delay** construct from RT-FRP instead of the **later** annotation in E-FRP. To do that, the syntax of E-FRP needs the following change:

$$\begin{array}{lrcl} \text{Event handlers} & H & ::= & \{I_j \Rightarrow e_j\} \\ \text{Stateful behaviors} & sf & ::= & \textbf{init}\ x = v\ \textbf{in}\ H \mid \textbf{delay}\ v\ e \end{array}$$

The rest of the syntax remains unchanged.

The semantics of **init-in** is the same as before except that we do not need to worry

about **later** reactions now. The semantics of **delay** can be given by:

$$\frac{}{P \vdash \textbf{delay}\ v\ e \xrightarrow{I}_{\searrow} v} \qquad \frac{P \vdash e \xrightarrow{I}_{\searrow} v'}{P \vdash \textbf{delay}\ v\ e \xrightarrow{I}_{\nearrow} \textbf{delay}\ v'\ e}$$

This **delay** construct basically acts the same as its RT-FRP counterpart. However, in the event-driven setting, it has a severe problem: a **delay** construct may behave differently in different contexts, thus breaking the modularity of the system.

To illustrate the problem, we consider the following example:

$$\mathsf{heartbeat} = \textbf{delay}\ 0\ (\mathsf{heartbeat} + 1).$$

This is the same $Heartbeat$ program we saw in Chapter 4, and it counts the number of event occurrences in the system.

The problem is, heartbeat counts *any* event occurrences. If it is used in a system having five events, it will count all of those *five* events; however, if we decide to add a module that uses a sixth events to the system, suddenly heartbeat will count the occurrences of *six* events — the meaning of this module is changed by the introduction of another module.

In general, when we design an event-driven system, we expect to be able to divide them to modules that have different functionalities, implement the modules independently, and then assemble these together. We also expect the function of a module is not affected by other modules, so we can reason about the system compositionally. This makes us believe the RT-FRP style delay is not a good construct for E-FRP.

5.7.2 A better delay

We notice that the shortcoming of **delay** can be remedied. In fact, the modularity problem comes from the fact that we do not specify which events affect the **delay** construct. If we do so, then adding events to the system will have no effect on the meaning a **delay** construct.

This new **delay** construct has the syntax:

$$\textbf{delay}\ v\ e\ \textbf{on}\ \{I_j\}$$

where $\{I_j\}$ is the set of events that affect this behavior.

The semantics is given by:

$$\frac{}{P \vdash \textbf{delay}\ v\ e\ \textbf{on}\ \{I_j\} \xrightarrow{I} v} \qquad \frac{I \in \{I_j\} \quad P \vdash e \xrightarrow{I} v'}{P \vdash \textbf{delay}\ v\ e\ \textbf{on}\ \{I_j\} \xrightarrow{I} \textbf{delay}\ v'\ e\ \textbf{on}\ \{I_j\}}$$

$$\frac{I \notin \{I_j\}}{P \vdash \textbf{delay}\ v\ e\ \textbf{on}\ \{I_j\} \xrightarrow{I} \textbf{delay}\ v\ e\ \textbf{on}\ \{I_j\}}$$

However, we point out that this new **delay** is no more expressive than **later**, as it can be expressed in terms of the latter. In general, the definition

$$x = \textbf{delay}\ v\ e\ \textbf{on}\ \{I_j\}$$

can be replaced by the following definition using **later**:

$$\begin{aligned} y &= e \\ x &= \textbf{init}\ \_ = v\ \textbf{in}\ \{I_j \Rightarrow y\ \textbf{later}\} \end{aligned}$$

where y is a fresh variable. Note that the translation works even when the original definition is recursive (i.e. when x occurs free in e).

Since **delay** can be defined using **later**, and our experience with micro-controllers shows that the **later** construct is often easier to use, E-FRP is designed to have **later** but no **delay**.

5.7.3 later as syntactic sugar

It might surprise the reader that the **later** annotation is not necessary even without **delay**! In fact, any E-FRP behavior definition that uses **later** can be rewritten without **later** by a simple translation, as long as the base language has pairs. More specifically, given

$$x = \textbf{init}\ y = v\ \textbf{in}\ \{I_j \Rightarrow e_j\} \uplus \{I_k \Rightarrow e_k\ \textbf{later}\},$$

we can rewrite it as

$$\begin{aligned} x' = \textbf{init}\ y' = (v, v)\ \textbf{in}\ &\{I_j \Rightarrow \textbf{let}\ \mathsf{z} = e_j[y := \mathsf{snd}(y')]\ \textbf{in}\ (\mathsf{z}, \mathsf{z})\} \uplus \\ &\{I_k \Rightarrow (\mathsf{snd}(y'), e_k[y := \mathsf{snd}(y')])\} \\ x\ = \textbf{case}\ x'\ \textbf{of}\ &(\mathsf{z}, \_) \Rightarrow \mathsf{z} \end{aligned}$$

where x' and y' are fresh variables, and $\mathsf{snd}(e)$ is syntactic sugar for

$$\textbf{case}\ e\ \textbf{of}\ (\_, \mathsf{z}) \Rightarrow \mathsf{z}.$$

Although **later** does not contribute to the expressiveness of E-FRP, having to revert to the above trick often significantly obscures the program, and therefore we decided to keep **later** in the language.

Chapter summary

We have presented a variant of RT-FRP called E-FRP that is suitable for event-driven real-time reactive systems, and have shown 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, and ICE code that can be used in the MindRover game. Our focus here is on the feasibility of implementing this language and using it to program real event-driven systems.

Chapter 6

Related work, future work, and conclusions

6.1 Related work

6.1.1 Hybrid automata

As hybrid systems are increasingly used in safety-critical applications, the concept of a hybrid automaton [22, 32] has been proposed as a formal model for reasoning about, and verifying properties of, such systems. A hybrid automaton can be viewed as a generalization of timed automata [2]. It has a finite number of *control modes*, in which the behavior of variables is governed by a set of differential equations. Discrete events trigger the system to jump from one mode to another. It has been shown that this concept is suitable for specifying a large variety of hybrid systems, and methods have been developed to verify interesting properties of certain categories of hybrid automata [1].

For a language to be used for hybrid reactive systems, the ability to express hybrid automata is important. As we have shown, H-FRP is both resource-bounded and capable of expressing hybrid automata directly using recursive integral behavior definitions and recursive switching. Under the conditions we have identified in Section 4.5, an H-FRP

program can implement a hybrid automaton as the maximum sampling interval goes to zero.

6.1.2 FRP

The work in this thesis is largely inspired by the Functional Reactive Programming (FRP) language, which is designed for modeling and programming hybrid reactive systems. FRP supports hybrid systems well by having both continuous-time-varying behaviors and discrete events as first-class values. In particular, hybrid automata can be expressed in FRP naturally.

FRP is an expressive language due to its extensive use of higher-order functions: signals can carry not only base type values like reals and Booleans, but also functions and even signals; common programming pattern can also be abstracted as higher-order functions. However, this flexibility is a double-edge sword, as it allows programs to use unlimited amount of space and time, and even diverge. The combination of higher-order functions and laziness makes reasoning about the cost of an FRP program extremely difficult in general. Thus FRP is not suitable for real-time systems. Nevertheless FRP has been successfully applied in many domains that do not have hard real-time requirements, including robotics [42] and animation [16].

RT-FRP is designed as a subset of FRP. While we keep RT-FRP small so that we can easily study its properties, we have shown that many FRP constructs can be defined in RT-FRP. We expect to translate a larger part of FRP to RT-FRP in the future.

Frappé [10] is an efficient implementation of a first-order subset of FRP. It translates FRP constructs to Java. A number of interesting evaluation strategies (called push, pull, and hybrid) have been explored in the context of Frappé and we are interested in formalizing these models and studying their properties. Frappé uses double-buffering in the translation, which is similar to our use of temporary variables in compiling E-FRP programs.

6.1.3 Synchronous data-flow languages

RT-FRP can be categorized as a synchronous data-flow language. Before RT-FRP, several other languages have been proposed around the synchronous data-flow notion of computation, including SIGNAL [18], LUSTRE [7], and ESTEREL [6, 4]. We will discuss the features of these languages and their connection with RT-FRP one by one.

Synchronous languages can be classified into *imperative* ones and *declarative* ones. The former defines a reactive system as a state-transition machine and specifies an explicit sequence of steps to follow to make a reaction. The latter defines a reactive system by specifying the relations between a set of signals using functions or constraints. Although an RT-FRP program describes a state machine, it does so by specifying the relations between the sub-components. Therefore we consider RT-FRP a declarative language.

In general, a system using recursive switching may use unbounded stacks. Hence, to achieve reactivity, none of the synchronous data-flow reactive languages has considered recursive switching. RT-FRP improves on them by allowing a restricted form of recursive switching.

RT-FRP clearly separates the base language and the reactive language and gives the user the choice on what is the best base language for the job at hand. This is not done in those synchronous reactive languages. We have found that such a separation makes RT-FRP more flexible, and helps us to clearly understand the two languages' individual contributions to the cost of execution.

ESTEREL

Reactive systems whose main purpose is to continuously produce output values from input ones are called *data-dominated*, and those focus on producing discrete output from input signals are said to be *control-dominated*. ESTEREL is a synchronous imperative language devoted to programming control-dominated software or hardware reactive systems [6].

Compilers are available that translate ESTEREL programs into finite state machines or digital circuits for embedded applications. These compilers perform heavy optimizations and can generate very efficient code, making ESTEREL a good choice for hardware design and performance-critical systems. In relation to our current work, a large effort has been made to develop a formal semantics for ESTEREL, including a constructive behavioral semantics, a constructive operational semantics, and an electrical semantics (in the form of digital circuits). These semantics are shown to correspond in a certain way, constrained only by a notion of stability.

ESTEREL does not handle data-processing well, and thus is not suitable for data-dominated systems. In contrast, RT-FRP supports both behaviors and events naturally, and can be used for both data-dominated and control-dominated systems.

SIGNAL

SIGNAL [18] is a synchronous declarative 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 RT-FRP program. Unlike in most other languages, a SIGNAL program is a set of constraints on signals, and the meaning of the program is the solution to the constraints. When two components are composed, the meaning of the composite component is the solution to the union of the constraints of the sub-components.

To verify a certain property of a program, the user can encode the property as SIGNAL equations. It could be then checked whether such equations are already implied by the program, or the equations may be simply added to the program to make sure the property is satisfied at run time. This is an important feature of SIGNAL.

Given a SIGNAL program (i.e. a set of constraints on signals), a solution for the constraints may not exist, in which case the system is undefined. It could also happen that more than one solutions can be found, in which case the system is non-deterministic. In general, it is mentally challenging for the programmer to find the solution for a program and determine whether it is unique. Therefore SIGNAL programs can be difficult to un-

derstand and maintain.

In contrast, RT-FRP specifies the relations between signals by functions rather than constraints. Hence RT-FRP programs are easier to understand than those written in SIGNAL. However, RT-FRP does not directly support verification of program properties. We plan to investigate the possibility of adding a SIGNAL-style verification mechanism to RT-FRP.

LUSTRE

LUSTRE is a language similar to SIGNAL, rooted again in the notion of a sequence. Unlike SIGNAL, LUSTRE programs define signals by functions and thus are deterministic. The LUSTRE formalism is very similar to temporal logics, so it can also be used for expressing program properties. A verification tool is implemented such that given a program with a single Boolean output, it checks that the output is never False. There is no fundamental difficulty in using the same approach for verifying properties of RT-FRP programs, and we are looking into such possibilities.

Synchronous Kahn networks

Caspi and Pouzet noticed the lack of expressiveness of synchronous data-flow reactive languages and proposed Synchronous Kahn networks [8], which extended these languages with recursive switching and higher-order programming, yielding a large increase in expressive power. While Synchronous Kahn networks still are synchronous, resource-boundedness is no longer guaranteed.

The expressiveness of RT-FRP lies somewhere between synchronous data-flow reactive languages and Synchronous Kahn networks: it supports recursive switching, yet at the same time manages to achieve a bound in space and response time using some syntactic restrictions and a type system.

6.1.4 SAFL

Mycroft and Sharp develop a statically allocated parallel functional language SAFL [37] for the specification of hardware. Their language allows recursion, but restricts recursive calls to tail calls. The purpose for such restriction is to be able to allocate all variables to fixed storage locations at compile time, which is essential for hardware design. As a result, the total space required for executing a program is $O(\textit{length of program})$.

As we are interested in using RT-FRP for real-time *software* systems, what is important is that there exists a bound for the space needed during program execution, not that all variables are allocated statically. Indeed, an RT-FRP program can allocate space dynamically, but the type system limits how much space it can consume at most. In the extreme case, a program can use exponential space with respect to the program size. This relaxation allows more programs to be expressed. In particular, it makes finite automata easy to encode.

SAFL uses an explicit notion of a syntactic context to specify what is a tail call, and restrict recursive functions. In our work, we have integrated this restriction into the type system. It will be interesting to see an integration is also possible in their setting.

6.1.5 Multi-stage programming

There are many connections between the semantics of RT-FRP and the semantics of multi-stage languages. Evaluating recursive **let-snapshot** declarations requires evaluation under a binder. This problem arises constantly in the context of multi-level and multi-stage languages [48]. Our approach to the treatment of this problem, namely, splitting the variable context into current variables and later variables in the type system, is inspired by the work on multi-stage languages [34, 50]. Our evaluation and updating functions are also analogous to the evaluation and rebuilding functions of multi-stage language [48].

6.1.6 SCR

The SCR (Software Cost Reduction) requirements method [20] 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* [21], 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* tool set to find errors in the program.

6.2 Future work

6.2.1 Clocked events

FRP and H-FRP have the lifting operators, which allow us to build a behavior by applying a function pointwise to a group of behaviors. In RT-FRP, such operators can be defined as syntactic sugar.

Sometimes, we know that certain events have the same occurrence pattern (i.e. one event occurs if and only if the other occurs), in which case we can apply a function to their occurrence values pointwise and get a composite event that has the same pattern. In other words, we want to write events in the form of

$$\textbf{lift}\ f\ ev_1\ ev_2$$

where ev_1 and ev_2 are known to be "in sync." For reliability of the system, we want to reject the above expression at compile time if ev_1 and ev_2 cannot be shown to always occur simultaneously.

In FRP, we call the occurrence pattern of an event its *clock*. Independent events are

deemed to have different clocks, while events that provably always occur together are said to share the same clock. For example, event mapping preserves the clock of its event operand. The clock is part of the type of an event. An event of type α with clock δ has type signature $\mathsf{E}\ \delta\ \alpha$. Hence events with different clocks have different type signatures.

In **lift** $f\ ev_1\ ev_2$, the **lift** operator has type

$$(\alpha \to \beta \to \gamma) \to \mathsf{E}\ \delta\ \alpha \to \mathsf{E}\ \delta\ \beta \to \mathsf{E}\ \delta\ \gamma.$$

The type ensures that the ev_1 and ev_2 have the same clock δ, and so does the result.

While in general whether two events have the same occurrence pattern is undecidable, we would like the clock inference to be as accurate as possible. In particular, we want the system to know that $e_1\ .|.\ e_2$ (an event that occurs whenever either e_1 or e_2 occurs) and $e_2\ .|.\ e_1$ have the same clock, while $e_1\ .|.\ e_2$ and $e_1\ .|.\ e_3$ do not (assuming that e_2 and e_3 have different clocks). Although it is not difficult to design a type system to convey this idea, we are not able to implement it for FRP since FRP is embedded in Haskell and therefore inherits Haskell's type system. As a result, the clocked events scheme in FRP is compromised.

For simplicity, RT-FRP as presented in this thesis does not support clocked events, which we expect to incorporate as future work. Since RT-FRP is a stand-alone language, we will be able to implement for it a type system that inferences clocks more accurately that the FRP type system does.

6.2.2 Arrow

When defining a large reactive system, it is often possible to divide the whole system into several inter-connected sub-systems that are driven by different types of stimuli, and define them individually. Such modular definitions are often easier to come up with and understand.

Yet, within an RT-FRP program, all components must have the same type of input

(i.e. the Input type is fixed across the program). This restriction is not compatible with the divide-and-conquer approach. Hence we would like to propose a method for combining RT-FRP machines with different stimulus types.

Courtney, Nilsson and Peterson suggest that Hughes' arrows [27] provide a natural mechanism for modeling signals that are explicitly parameterized by an input type. The benefits of the arrow framework include better modularity and reduced space leak. They have designed a version of FRP [38] in this framework, and the utility of this language has been confirmed in the setting of robotics [41]. We expect that this approach can be used as a basis for a language for combining RT-FRP machines.

6.2.3 Better models for real-time systems

In this thesis, we have chosen to focus on the issue of bounded resources in the presence of recursion. Ultimately, however, we are interested in more sophisticated models for real-time systems, where resources are allocated according to their priority (see Kieburtz [30] for a nice account from the Haskell point of view). SIGNAL, LUSTRE, and synchronous Kahn networks [8] account for this via a clock calculus, which allows different parts of the system to run at different frequencies. While this technique may apply directly to RT-FRP, this still remains to be established.

Arrows also provide a way for combining computation performed at different frequencies, through its ArrowChoice operator [40]. It remains to be investigated whether this or the clock calculus is a better solution.

6.2.4 Parallel execution

The operational semantics of RT-FRP relegates all actual computations to the base language. Since we require the base language to be pure, the order in which expressions are evaluated does not affect the result of an RT-FRP program. In particular, independent expressions can be evaluated concurrently. As Hammond [19] observes, "it is almost

embarrassingly easy to partition a program written in a strict [functional] language [into parallel threads].” We expect to be able to develop such a partitioning strategy for RT-FRP programs.

6.3 Conclusions

We have designed a language RT-FRP for real-time reactive systems. Programs written in this language are guaranteed to consume bounded space and time for each reaction. RT-FRP improves on synchronous data-flow reactive languages by allowing recursive switching to be expressed. H-FRP and E-FRP, two variants of RT-FRP, are developed to better suit the need for programming hybrid systems and event-driven systems. These languages improve on traditional languages for their respective domains.

Our theoretical results include formal proofs for the resource-boundedness of RT-FRP, a set of conditions under which the discrete execution of an H-FRP program converges to its continuous semantics, and a provably correct compiler for E-FRP.

The applicability of our approach has been tested in the context of micro-controllers and computer-simulated robotics.

Appendix A

Proof of theorems

A.1 Theorems in Chapter 3

Lemma 3.4. $|s|$ is defined for all s.

Proof. By induction on s.

1. $s \equiv$ **input**, **time**, or **ext** e. By the definition of $|-|$, we have that $|s|$ is defined.
2. $s \equiv$ **delay** $e\ s'$. By the induction hypothesis, $|s'|$ is defined. Therefore $|s| = 2 + |s'|$ is defined.
3. $s \equiv$ **let snapshot** $x \leftarrow s_1$ **in** s_2. By the induction hypothesis, $|s_1|$ and $|s_2|$ are defined. Therefore $|s| = 2 + |s_1| + |s_2|$ is defined.
4. $s \equiv$ **let signal** $\{\, z_j(x_j) = u_j \,\}^{j \in \mathbb{N}_n}$ **in** s'. By the induction hypothesis, $|s'|$ is defined, and for all $j \in \mathbb{N}_n$, $|u_j|$ is defined. Therefore $|s| = 1 + 2n + \Sigma_{j=1}^{n} |u_j| + |s'|$ is defined.
5. $s \equiv s'$ **until** $\langle\, s_j \Rightarrow z_j \,\rangle^{j \in \mathbb{N}_n}$. By the induction hypothesis, $|s'|$ is defined, and for all $j \in \mathbb{N}_n$, $|s_j|$ is defined. Therefore $|s| = 1 + n + |s'| + \Sigma_{j=1}^{n} |s_j|$ is defined.

And we are done. □

Lemma 3.5. For all term s and integer m, $\|s\|_m$ is defined.

Proof. By induction on s. Given a term s, we assume that for all sub-term s' in s, $\|s'\|_m$ is defined for all integer m.

1. $s \equiv$ **input**, **time**, or **ext** e. By the definition of $\|-\|_m$, $\|s\|_m$ is defined.

2. $s \equiv$ **delay** $e\ s'$. By the induction hypothesis, $\|s'\|_m$ is defined. Therefore $\|s\|_m = 2 + \|s'\|_m$ is defined.

3. $s \equiv$ **let snapshot** $x \leftarrow s_1$ **in** s_2. By the induction hypothesis, $\|s_1\|_m$ and $\|s_2\|_m$ are defined. Therefore $\|s\|_m = 2 + \|s_1\|_m + \|s_2\|_m$ is defined.

4. $s \equiv$ **let signal** $\{\, z_j(x_j) = u_j \,\}^{j \in \mathbb{N}_n}$ **in** s'. By the induction hypothesis, for all $j \in \mathbb{N}_n$, $\|u_j\|_0$ is defined. Therefore

$$k = \left\| \{ z_j \mapsto \lambda x_j.u_j \}^{j \in \mathbb{N}_n} \right\| = \max\left(\{0\} \cup \left\{ \|u_j\|_0 \right\}^{j \in \mathbb{N}_n} \right)$$

   is defined. Hence $\max\{k, m\}$ is defined, and by the induction hypothesis $\|s'\|_{\max\{k,m\}}$ is defined. Therefore, $\|s\|_m = 1 + 2n + \Sigma_{j=1}^{n} |u_j| + \|s'\|_{\max\{k,m\}}$ is defined.

5. $s \equiv s'$ **until** $\langle\, s_j \Rightarrow z_j \,\rangle^{j \in \mathbb{N}_n}$. By the induction hypothesis, $\|s'\|_0$ is defined, and for all $j \in \mathbb{N}_n$, $\|s_j\|_0$ is defined. Therefore $\|s\|_m = 1 + n + \|s'\|_0 + \Sigma_{j=1}^{n} \|s_j\|_0$ is defined.

And we are done. □

Lemma 3.6. For all term s and integer m, $|s| \leq \|s\|_m$.

Proof. By induction on s. Given a term s, we assume that for each sub-term s' of s, $|s'| \leq \|s'\|_m$ for all integer m.

1. $s \equiv$ **input**, **time**, or **ext** e. By the definition of $|-|$ and $\|-\|_m$, we have $|s| = \|s\|_m$.

2. $s \equiv$ **delay** $e\ s'$.

$$\begin{aligned} |s| &= 2 + |s'| \\ &\leq 2 + \|s'\|_m \quad \text{(Induction hypothesis)} \\ &= \|s\|_m \end{aligned}$$

3. $s \equiv$ **let snapshot** $x \leftarrow s_1$ **in** s_2.

$$\begin{aligned} |s| &= 2 + |s_1| + |s_2| \\ &\leq 2 + \|s_1\|_m + \|s_2\|_m \quad \text{(Induction hypothesis)} \\ &= \|s\|_m \end{aligned}$$

4. $s \equiv$ **let signal** $\{\, z_j(x_j) = u_j \,\}^{j \in \mathbb{N}_n}$ **in** s'.

$$\begin{aligned} |s| &= 1 + 2n + \Sigma_{j=1}^{n} |u_j| + |s'| \\ &\leq 1 + 2n + \Sigma_{j=1}^{n} |u_j| + \|s'\|_{\max\{\|\{z_j \mapsto \lambda x_j.u_j\}^{j \in \mathbb{N}_n}\|, m\}} \quad \text{(Induction hypothesis)} \\ &= \|s\|_m \end{aligned}$$

5. $s \equiv s'$ **until** $\langle\, s_j \Rightarrow z_j \,\rangle^{j \in \mathbb{N}_n}$.

$$\begin{aligned} |s| &= 1 + n + |s'| + \Sigma_{j=1}^{n} |s_j| \\ &\leq 1 + n + \|s'\|_0 + \Sigma_{j=1}^{n} \|s_j\|_0 \quad \text{(Induction hypothesis)} \\ &\leq \max\{\, 1 + n + \|s'\|_0 + \Sigma_{j=1}^{n} \|s_j\|_0, m \,\} \\ &= \|s\|_m \end{aligned}$$

And we are done. □

Lemma 3.7. $\mathcal{X}; \mathcal{Z} \vdash s \xrightarrow{t,i} s'$ implies $\|s'\|_{\|\mathcal{Z}\|} \leq \|s\|_{\|\mathcal{Z}\|}$.

Proof. The proof is by induction on the derivation for $\mathcal{X}; \mathcal{Z} \vdash s \xrightarrow{t,i} s'$. In what follows, we let $m = \|\mathcal{Z}\|$.

1. The last rule used in the derivation is u1, u2, or u3. In this case $s \equiv s'$. Hence $\|s'\|_m \leq \|s\|_m$.

2. The last rule in the derivation is u4. Then $s \equiv$ **delay** $v\ s_1$, $s' \equiv$ **delay** $v'\ s_1'$, and

$\mathcal{X};\ \mathcal{Z} \vdash s_1 \xrightarrow{t,i} s_1'$. By induction hypothesis,

$$\left\|s'\right\|_m = 2 + \left\|s_1'\right\|_m \leq 2 + \|s_1\|_m = \|s\|_m.$$

3. The last rule is u5. We have that $s \equiv$ **let snapshot** $x \leftarrow s_1$ **in** s_2, $s' \equiv$ **let snapshot** $x \leftarrow s_1'$ **in** s_2', $\mathcal{X} \vdash s_1 \overset{t,i}{\hookrightarrow} v_1$, $\mathcal{X} \cup \{x \mapsto v_1\}; \mathcal{Z} \vdash s_1 \xrightarrow{t,i} s_1'$, and $\mathcal{X} \cup \{x \mapsto v_1\}; \mathcal{Z} \vdash s_2 \xrightarrow{t,i} s_2'$. Therefore

$$\begin{aligned} \|s'\|_m &= 2 + \|s_1'\|_m + \|s_2'\|_m \\ &\leq 2 + \|s_1\|_m + \|s_2\|_m \quad \text{(Induction hypothesis)} \\ &= \|s\|_m \end{aligned}$$

4. The last rule is u6. We know that

$$\begin{gathered} s \equiv \textbf{let signal } \{\, z_j(x_j) = u_j \,\} \textbf{ in } s_1 \\ s' \equiv \textbf{let signal } \{\, z_j(x_j) = u_j \,\} \textbf{ in } s_1' \\ \text{and } \mathcal{X};\ \mathcal{Z} \cup \{z_j \mapsto \lambda x_j.u_j\} \vdash s_1 \xrightarrow{t,i} s_1' \end{gathered}$$

Note that

$$\begin{aligned} & \|\mathcal{Z} \cup \{z_j \mapsto \lambda x_j.u_j\}\| \\ = \ & \max\{\max\{\|u_j\|_0\},\ m\} \\ = \ & \max\{\|\{z_j \mapsto \lambda x_j.u_j\}\|,\ m\} \end{aligned}$$

Hence,

$$\begin{aligned} & \|s'\|_m \\ = \ & 1 + 2n + \Sigma_{j=1}^n |u_j| + \|s_1'\|_{\max\{\|\{z_j \mapsto \lambda x_j.u_j\}\|,\ m\}} \\ \leq \ & 1 + 2n + \Sigma_{j=1}^n |u_j| + \|s_1\|_{\max\{\|\{z_j \mapsto \lambda x_j.u_j\}\|,\ m\}} \quad \text{(Induction hypothesis)} \\ = \ & \|s\|_m \end{aligned}$$

5. The last rule is u7. In this case,

$$s \equiv s_0 \textbf{ until } \langle\, s_j \Rightarrow z_j \,\rangle^{j \in \mathbb{N}_n},$$

and $s' \equiv u[x := v]$, where $(z \mapsto \lambda x.u) \in \mathcal{Z}$ for some z and x.

$$\begin{aligned}
& \|s'\|_m \\
= \;& \|u\|_m \\
= \;& \max\{\, 1 + n' + \|s'_0\|_0 + \Sigma_{j=1}^{n'} \left\| s'_j \right\|_0, m \,\} \\
& \quad \text{where } u \equiv s'_0 \textbf{ until } \langle s'_j \Rightarrow z'_j \rangle^{j \in \mathbb{N}_{n'}} \\
= \;& \max\{\, 1 + n' + \|s'_0\|_0 + \Sigma_{j=1}^{n'} \left\| s'_j \right\|_0, 0, m \,\} \\
= \;& \max\{\|u\|_0, m\} \\
= \;& m \quad ((z \mapsto \lambda x.u) \in \mathcal{Z} \text{ and } m = \|\mathcal{Z}\|) \\
\leq \;& \max\left\{1 + n + \|s_0\|_0 + \Sigma_{j=1}^{n} \|s_j\|_0, m\right\} \\
= \;& \|s\|_m.
\end{aligned}$$

6. The last rule is u8. We have that $s \equiv s_0 \textbf{ until } \langle\, s_j \Rightarrow z_j \,\rangle^{j \in \mathbb{N}_n}$, $s' \equiv s'_0 \textbf{ until } \langle\, s'_j \Rightarrow z_j \,\rangle^{j \in \mathbb{N}_n}$, and $\left\{ \mathcal{X}; \emptyset \vdash s_j \xrightarrow{t,i} s'_j \right\}^{j \in \{0..n\}}$. Therefore

$$\begin{aligned}
\|s'\|_m &= \max\left\{1 + n + \|s'_0\|_0 + \Sigma_{j=1}^{n} \left\| s'_j \right\|_0, m\right\} \\
&\leq \left\{1 + n + \|s_0\|_0 + \Sigma_{j=1}^{n} \|s_j\|_0, m\right\} \quad \text{(Induction hypothesis)} \\
&= \|s\|_m.
\end{aligned}$$

And we are done. □

Lemma 3.8. For any term s and integer $m \geq 0$, we have that $\|s\|_m \leq \max\{1, m\} \cdot 2^{|s|}$.

Proof. By induction on s. We assume that for any sub-term s' of s, we have $\|s'\|_m \leq \max\{1, m\} \cdot 2^{|s'|}$ for all integer $m \geq 0$.

1. $s \equiv$ **input** or **time**. We have

$$\begin{aligned} \|s\|_m &= 1 \\ &\leq \max\{1, m\} \cdot 2^1 \\ &= \max\{1, m\} \cdot 2^{|s|}. \end{aligned}$$

2. $s \equiv$ **ext** e. We have

$$\begin{aligned} \|s\|_m &= 2 \\ &\leq \max\{1, m\} \cdot 2^2 \\ &= \max\{1, m\} \cdot 2^{|s|}. \end{aligned}$$

3. $s \equiv$ **delay** $e\ s'$. It follows that

$$\begin{aligned} \|s\|_m &= 2 + \|s'\|_m \\ &\leq 2 + \max\{1, m\} \cdot 2^{|s'|} \\ &\leq \max\{1, m\} \cdot 2^{2+|s'|} \\ &= \max\{1, m\} \cdot 2^{|s|}. \end{aligned}$$

4. $s \equiv$ **let snapshot** $x \leftarrow s_1$ **in** s_2**.** We have

$$\begin{aligned} \|s\|_m &= 2 + \|s_1\|_m + \|s_2\|_m \\ &\leq 2 + \max\{1, m\} \cdot 2^{|s_1|} + \max\{1, m\} \cdot 2^{|s_2|} \\ &\leq \max\{1, m\} \cdot 2^{2+|s_1|+|s_2|} \\ &= \max\{1, m\} \cdot 2^{|s|}. \end{aligned}$$

5. $s \equiv$ **let signal** $\{\, z_j(x_j) = u_j \,\}^{j \in \mathbb{N}_n}$ **in** s'. We have

$$\begin{array}{rcl}
\|s\|_m & = & 1 + 2n + \Sigma_{j=1}^{n} |u_j| + \|s'\|_{\max\{\|\{z_j \mapsto \lambda x_j.u_j\}^{j \in \mathbb{N}_n}\|,\, m\}} \\
& = & 1 + 2n + \Sigma_{j=1}^{n} |u_j| + \|s'\|_{\max\{k,\, m\}} \\
& & \quad \text{where } k = \left\| \{z_j \mapsto \lambda x_j.u_j\}^{j \in \mathbb{N}_n} \right\| = \max\left(\{0\} \cup \left\{ \|u_j\|_0 \right\}^{j \in \mathbb{N}_n} \right) \\
& \leq & 1 + 2n + \Sigma_{j=1}^{n} |u_j| + \max\left(\left\{ 2^{|u_j|} \right\}^{j \in \mathbb{N}_n} \cup \{m\} \right) \cdot 2^{|s'|} \\
& & \quad \text{(Induction hypothesis)} \qquad\qquad (*)
\end{array}$$

We want to prove that $(*) \leq (\#)$, where $(\#) = \max\{1, m\} \cdot 2^{1+2n+\Sigma_{j=1}^{n}|u_j|+|s'|} = \max\{1, m\} \cdot 2^{|s|}$. There are three cases:

(a) $n = 0$. We have

$$\begin{array}{rcl}
(*) & = & 1 + m \cdot 2^{|s'|} \\
& \leq & 2^{|s'|} + m \cdot 2^{|s'|} \\
& \leq & (\max\{1, m\} + \max\{1, m\}) \cdot 2^{|s'|} \\
& = & (\#).
\end{array}$$

(b) $n \geq 1$ and $m \leq \max\left\{2^{|u_j|}\right\}^{j \in \mathbb{N}_n}$. We have

$$\begin{array}{rcl}
(*) & = & 1 + 2n + \Sigma_{j=1}^{n} |u_j| + 2^{\max\{|u_j|\}^{j \in \mathbb{N}_n} + |s'|} \\
& \leq & 2^{1+2n+\Sigma_{j=1}^{n}|u_j|} + 2^{\max\{|u_j|\}^{j \in \mathbb{N}_n} + |s'|} \\
& \leq & 2^{1+2n+\Sigma_{j=1}^{n}|u_j|} + 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} \\
& \leq & 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} + 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} \\
& \leq & 2^{1+2n+\Sigma_{j=1}^{n}|u_j|+|s'|} \\
& \leq & (\#).
\end{array}$$

(c) $n \geq 1$ and $m > \max\left\{2^{|u_j|}\right\}^{j\in\mathbb{N}_n}$. Hence $m > 2$. We have

$$\begin{aligned}
(*) &= 1 + 2n + \Sigma_{j=1}^{n}|u_j| + m \cdot 2^{|s'|} \\
&\leq 1 + 2n + \Sigma_{j=1}^{n}|u_j| + m \cdot 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} \\
&\leq 2^{1+2n+\Sigma_{j=1}^{n}|u_j|} + m \cdot 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} \\
&< m \cdot 2^{1+2n+\Sigma_{j=1}^{n}|u_j|} + m \cdot 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} \\
&\leq m \cdot 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} + m \cdot 2^{2n+\Sigma_{j=1}^{n}|u_j|+|s'|} \\
&= (\#).
\end{aligned}$$

6. $s \equiv s'$ **until** $\langle\, s_j \Rightarrow z_j \,\rangle^{j\in\mathbb{N}_n}$. We have

$\|s\|_m = \max\left\{1 + n + \|s'\|_0 + \Sigma_{j=1}^{n}\|s_j\|_0, m\right\}$. We want to prove that

$$\|s\|_m \leq \max\{1, m\} \cdot 2^{|s|}.$$

It suffices to prove that

$$1 + n + \left\|s'\right\|_0 + \Sigma_{j=1}^{n}\|s_j\|_0 \leq \max\{1, m\} \cdot 2^{|s|}$$

and

$$m \leq \max\{1, m\} \cdot 2^{|s|}.$$

The latter is trivial. Now we try to prove the former:

$$\begin{aligned}
1 + n + \|s'\|_0 + \Sigma_{j=1}^{n}\|s_j\|_0 &\leq 1 + n + 2^{|s'|} + \Sigma_{j=1}^{n}2^{|s_j|} \quad \text{(Induction hypothesis)} \\
&\leq 2^{1+n+|s'|+\Sigma_{j=1}^{n}|s_j|} \quad \text{(See comments below.)} \\
&= 2^{|s|} \\
&\leq \max\{1, m\} \cdot 2^{|s|}.
\end{aligned}$$

The step

$$1 + n + 2^{|s'|} + \Sigma_{j=1}^{n} 2^{|s_j|} \leq 2^{1+n+|s'|+\Sigma_{j=1}^{n}|s_j|}$$

is proved by noticing that there are two cases:

(a) $n = 0$. We have

$$\begin{aligned} 1 + n + 2^{|s'|} + \Sigma_{j=1}^{n} 2^{|s_j|} &= 1 + 2^{|s'|} \\ &\leq 2^{1+|s'|} \\ &= 2^{1+n+|s'|+\Sigma_{j=1}^{n}|s_j|}. \end{aligned}$$

(b) $n \geq 1$. In this case we have

$$\begin{aligned} 1 + n + 2^{|s'|} + \Sigma_{j=1}^{n} 2^{|s_j|} &\leq 2^{1+n} + 2^{|s'|} + \Sigma_{j=1}^{n} 2^{|s_j|} \\ &\leq 2^{1+n} + 2^{|s'|} + 2^{\Sigma_{j=1}^{n}|s_j|} \\ &\leq 2^{1+n+|s'|+\Sigma_{j=1}^{n}|s_j|}. \end{aligned}$$

□

A.2 Theorems in Chapter 4

To ease the proof of the theorems, first we prove the following causality lemma, which states that the responses of the system has made so far do not depend on the future:

Lemma A.1 (Causality). Given an H-FRP term $\tau \in B \cup EV$ and an environment $\mathcal{B}$ where $\vdash \tau; \mathcal{B}$, for any time sequence $P = \langle t_1, t_2, \cdots, t_n \rangle$, let $\langle v_1, v_2, \cdots, v_n \rangle = \widetilde{\text{trace}}[\![(]\!]\tau)_{\mathcal{B}}\ P$, we have that

$$\forall k \in \mathbb{N}_n .\ \widetilde{\text{trace}}[\![\tau]\!]_{\mathcal{B}}\ \langle t_1, \cdots, t_k \rangle = \langle v_1, \cdots, v_k \rangle.$$

Proof. By the definition of $\widetilde{\text{trace}}[\![-]\!]$. □

Theorem 4.2 (Time). $\vdash$ **time**; $\mathcal{B} \implies \widetilde{\text{at}}[\![\textbf{time}]\!]_{\mathcal{B}}\ P^t \rightarrowtail t\big|_{t_0}^{\infty}$.

Proof. Following the operational semantics, we have

$$\widetilde{\text{trace}}[\![\textbf{time}]\!]_{\mathcal{B}}\ P^t \equiv P^t.$$

Therefore

$$\widetilde{\text{at}}[\![\textbf{time}]\!]_{\mathcal{B}}\ P^t = t,$$

and it follows by the definition of uniform convergence that

$$\widetilde{\text{at}}[\![\textbf{time}]\!]_{\mathcal{B}}\ P^t \rightarrowtail t\big|_{t_0}^{\infty}.$$

□

Theorem 4.3 (Input). $\vdash$ **input**; $\mathcal{B} \implies \widetilde{\text{at}}[\![\textbf{input}]\!]_{\mathcal{B}}\ P^t \rightarrowtail \text{input}(t)\big|_{t_0}^{\infty}$.

Proof. Following the operational semantics, we have

$$\widetilde{\text{trace}}[\![\textbf{input}]\!]_{\mathcal{B}}\ \langle t_1, \cdots, t_n\rangle \equiv \langle \text{input}(t_1), \cdots, \text{input}(t_n)\rangle.$$

Therefore

$$\widetilde{\text{at}}[\![\textbf{input}]\!]_{\mathcal{B}}\ P^t = \text{input}(t),$$

and it follows by the definition of uniform convergence that

$$\widetilde{\text{at}}[\![\textbf{input}]\!]_{\mathcal{B}}\ P^t \rightarrowtail \text{input}(t)\big|_{t_0}^{\infty}.$$

□

Theorem 4.4 (Lift). Given $b \equiv$ **lift** $(\lambda(x_1, \cdots, x_n).e)\ b_1\ \cdots\ b_n$, if we have:

1. $\vdash b$; $\mathcal{B}$,
2. $\lambda(y_1, \cdots, y_n).$**eval**$[\![e]\!]_{\{x_j \mapsto y_j\}}$ is uniformly continuous, and

3. $\forall j \in \mathbb{N}_n.\ \widetilde{\text{at}}[\![b_j]\!]_{\mathcal{B}}\ P^t \rightarrowtail f_j(t)\Big|_{t_0}^{T} - S_j$,

then

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail \textbf{eval}[\![e]\!]_{\{x_j \mapsto f_j(t)\}}\Big|_{t_0}^{T} - \bigcup_{j=1}^{n} S_j.$$

Proof. Let $F_j = \widetilde{\text{at}}[\![b_j]\!]_{\mathcal{B}}$, $g(y_1, \cdots, y_n) = \textbf{eval}[\![e]\!]_{\{x_j \mapsto y_j\}}$, and $H = \widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}$. There are two cases to consider:

1. $n = 0$. $H(P^t) = \textbf{eval}[\![e]\!]_{\emptyset}$. It follows trivially that

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail \textbf{eval}[\![e]\!]_{\{x_j \mapsto f_j(t)\}}\Big|_{t_0}^{T} - \bigcup_{j=1}^{n} S_j.$$

2. $n > 0$. For any given $\epsilon > 0$, since g is uniformly continuous, there is a $\delta \in (0, \epsilon)$ such that

$$\big|\, v - v' \,\big| < \delta \implies \big|\, g(v) - g(v') \,\big| < \epsilon.$$

For any $j \in \mathbb{N}_n$, since $F_j(P^t) \rightarrowtail f_j(t)\Big|_{t_0}^{T} - S_j$, we have that there is $\gamma_j > 0$ such that $t \in [0,T] - [S_j]_{\delta/n}$ and $\big|P^t\big|_{t_0} < \gamma_j$ imply $\big|\, F_j(P^t) - f_j(t) \,\big| < \delta/n$. Let $\gamma = \min\{\gamma_j\}$, then for any $j \in \mathbb{N}_n$, $t \in [0,T] - [S_j]_{\delta/n}$ and $\big|P^t\big|_{t_0} < \gamma$ imply $\big|\, F_j(P^t) - f_j(t) \,\big| < \delta/n$. Since $\delta/n < \epsilon/n \leq \epsilon$, we have $\forall j \in \mathbb{N}_n.\ \Big([0,T] - \Big[\bigcup_{j=1}^{n} S_j\Big]_{\epsilon}\Big) \subset \Big([0,T] - [S_j]_{\delta/n}\Big)$, and therefore $t \in [0,T] - \Big[\bigcup_{j=1}^{n} S_j\Big]_{\epsilon}$ and $\big|P^t\big|_{t_0} < \gamma$ imply that

$$\big|\, (F_1(P^t), \cdots, F_n(P^t)) - (f_1(t), \cdots, f_n(t)) \,\big| < \delta.$$

This in turn leads to $\big|\, g(F_1(P^t), \cdots, F_n(P^t)) - g(f_1(t), \cdots, f_n(t)) \,\big| < \epsilon$.

By the operational semantics of **lift**, we have $H(P^t) = g(F_1(P^t), \cdots, F_n(P^t))$. Therefore we have established that

$$\forall \epsilon > 0.\ \exists \gamma > 0.\ (t \in [0,T] - \Big[\bigcup_{j=1}^{n} S_j\Big]_{\epsilon} \text{ and } \big|P^t\big|_{t_0} < \gamma) \implies \big|\, H(P^t) - g(f_1(t), \cdots, f_n(t)) \,\big| < \epsilon.$$

Hence

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail \textbf{eval}[\![e]\!]_{\{x_j \mapsto f_j(t)\}}\Big|_{t_0}^{T} - \bigcup_{j=1}^{n} S_j.$$

□

The following two lemmas are needed for proving Theorem 4.5. Lemma A.2 says that if f is integrable on a closed interval I, then it is also integrable on sub-intervals of I. Lemma A.3 establishes a further property: if f if integrable on a closed interval I, then the sum-of-the-areas *uniformly* converge to the integral of f on sub-intervals of I.

Lemma A.2. For all $a \leq c \leq d \leq b$, we have

$$\int_a^b f(\tau)\mathrm{d}\tau \neq \perp \implies \int_c^d f(\tau)\mathrm{d}\tau \neq \perp.$$

Lemma A.3. Given a function f and real numbers $a \leq b$, if $\int_a^b f(\tau)\mathrm{d}\tau \neq \perp$, we have that for all $\epsilon > 0$, there is $\delta > 0$, such that

$$\forall [c,d] \subseteq [a,b] \text{ and } P = \langle t_1 = c, \cdots, t_n = d\rangle.\ |P| < \delta \implies$$
$$\left|\Sigma_{j=1}^{n-1} f(\xi_j)\Delta t_j - \int_c^d f(\tau)\mathrm{d}\tau\right| \leq \epsilon$$

where ξ_j is arbitrarily chosen from the interval $[t_j, t_{j+1}]$.

*Proof **(Lemma A.2)**.* Let $F = \int_a^b f(\tau)\mathrm{d}\tau$.

For any given $\epsilon_1 > 0$, since $F \neq \perp$, by the definition of Riemann integral, we know that there is $\delta_1 > 0$ such that for all $P = \langle t_1, \cdots, t_n\rangle$ where $a = t_1 \leq t_2 \leq \cdots \leq t_n = b$, if $|P| < \delta_1$, we have

$$\left|\Sigma_{j=1}^{n-1} f(\xi_j)\Delta t_j - F\right| \leq \epsilon_1$$

where ξ_j is arbitrarily chosen from the interval $[t_j, t_{j+1}]$.

We can choose P such that it contains the point c and d, i.e. there are $m, k \in \mathbb{N}_n$ giving

us $t_m = c$ and $t_k = d$. Obviously, $1 \leq m \leq k \leq n$. Then we rewrite the above formula as

$$\left|\Sigma_{j=1}^{m-1} f(\xi_j)\Delta t_j + \Sigma_{j=m}^{k-1} f(\xi_j)\Delta t_j + \Sigma_{j=k}^{n-1} f(\xi_j)\Delta t_j - F\right| \leq \epsilon_1.$$

Let $A = \Sigma_{j=1}^{m-1} f(\xi_j)\Delta t_j + \Sigma_{j=k}^{n-1} f(\xi_j)\Delta t_j - F$ and $B = \Sigma_{j=m}^{k-1} f(\xi_j)\Delta t_j$. The above can be further rewritten as

$$|A + B| \leq \epsilon_1.$$

For any non-empty non-descending sequence $Q = \langle \tau_1, \cdots, \tau_l \rangle$, where $\tau_1 = c$ and $\tau_l = d$, and $|Q| < \delta_1$, it is obvious that the norm of $\langle t_1, \cdots, t_{m-1} \rangle \frown Q \frown \langle t_k, \cdots, t_n \rangle$ is also less than δ_1. Thus we have

$$\left|A + B'\right| \leq \epsilon_1$$

where $B' = \Sigma_{j=1}^{l-1} f(\omega_j)\Delta\tau_j$ and ω_j is arbitrarily chosen from $[\tau_j, \tau_{j+1}]$.

Then we can see that the difference between B' and B is bounded by $2\epsilon_1$:

$$\left|B' - B\right| = \left|(A + B') - (A + B)\right| \leq \left|A + B'\right| + |A + B| \leq 2\epsilon_1.$$

In other words, letting interval $I_1 = [B - 2\epsilon_1, B + 2\epsilon_1]$, for any non-empty non-descending sequence $Q = \langle c, \cdots, d \rangle$, where $|Q| < \delta_1$, we have that

$$g(Q) \in I_1$$

where $g(Q)$ is the notation we adopt from now on [1] for

$\Sigma_{j=1}^{l-1} f(\omega_j)\Delta\tau_j$ where $\langle \tau_1, \cdots, \tau_l \rangle \equiv Q$ and ω_j is arbitrarily chosen from $[\tau_j, \tau_{j+1}]$.

Next, we pick an $\epsilon_2 > 0$. By the same reasoning, we know that there is $\delta_2 \in (0, \delta_1)$, such that for any non-empty non-descending sequence $Q = \langle c, \cdots, d \rangle$, where $|Q| < \delta_2$,

[1] The reader must remember that g is *not* to be treated as a function, since it is not. Instead, we should regard $g(Q)$ as an (atomic) shorthand for the above expression.

we have that

$$g(Q) \in I_2$$

where we arbitrarily choose a Q_2 satisfying the above requirements and let $B_2 = g(Q_2)$, and $I_2 = [B_2 - 2\epsilon_2, B_2 + 2\epsilon_2]$.

Obviously, $B_2 \in I_2$. At the same time, since $|Q_2| < \delta_1$, we know that $B_2 = g(Q_2) \in I_1$.

Given an (infinite) series of positive real numbers $\epsilon_1, \epsilon_2, \cdots$, by repeating the above reasoning, we can establish that for all $j \geq 2$, we can find a $\delta_j \in (0, \delta_{j-1})$ (depending only on ϵ_j), such that for all $Q \equiv \langle c, \cdots, d\rangle$ where $|Q| < \delta_j$, we have

$$g(Q) \in I_j$$

where we arbitrarily choose a Q_j satisfying the above requirements and let $B_j = g(Q_j)$, and $I_j = [B_j - 2\epsilon_j, B_j + 2\epsilon_j]$. Besides, we know $B_j \in I_1 \cap \cdots \cap I_j$.

Let $J_j = I_1 \cap \cdots \cap I_j$ for all $j \in \mathbb{N}$. Since $B_j \in J_j$, we know J_j is not empty. Furthermore, for all $j \in \mathbb{N}$, we have J_j is a closed interval enclosing J_{j+1}.

Since $J_j \subseteq I_j$ and the length of I_j is $4\epsilon_j$, we know the length of J_j is no more than $4\epsilon_j$.

By choosing the series $\epsilon_1, \epsilon_2, \ldots$ such that ϵ_j tend to zero (for example, $\epsilon_j = 2^{-j}$), we can make the lengths of J_j tend to zero. In this case, $J_1, J_2, \ldots$ are called a *nested sequence of intervals* [9, page 8].

Now recall the *axiom of continuity* (also known as the *axiom of nested intervals*) for real numbers [9, page 8,95], which says "if $J_1, J_2, \ldots$ form a nested sequence of intervals, there is a unique point $x \in \mathbb{R}$ contained in all J_j." Therefore, there is a unique $x \in R$ contained in all J_j. By the definition of Riemann integral, $\int_c^d f(\tau)\mathrm{d}\tau = x \neq \perp$. And we are done. □

Proof ***(Lemma A.3)***. The proof is easy once we understand the proof for Lemma A.2.

Given $\epsilon > 0$, let $\gamma = \epsilon/4$. By the same reasoning as in the proof for Lemma A.2, we know there is $\delta > 0$ (depending only on γ), such that for all non-empty non-descending

sequence $P = \langle t_1, \cdots, t_n \rangle$ where $c = t_1 \leq t_2 \leq \cdots \leq t_n = d$,

$$|P| < \delta \implies \left| \Sigma_{j=1}^{n-1} f(\xi_j) \Delta t_j - \int_c^d f(\tau) \mathrm{d}\tau \right| \leq 4\gamma = \epsilon$$

where ξ_j is arbitrarily chosen from $[t_j, t_{j+1}]$. And we are done. □

Theorem 4.5 (Integral). If

1. $\vdash$ **integral** b; $\mathcal{B}$,
2. $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T} - S$ where S is finite, and
3. there is $T_0 \lessdot t_0$ such that
   (a) for all time sequence $P^t = \langle t_1, \cdots, t \rangle$ where $T_0 \leq t_1 \leq t \leq T$, $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t$ is bounded,
   (b) $f(t)$ is bounded on $[T_0, T]$, and
   (c) $\int_{T_0}^{T} f(t)\mathrm{d}t \neq \perp$,

then we have

$$\widetilde{\text{at}}[\![\textbf{integral}\ b]\!]_{\mathcal{B}}\ P^t \rightarrowtail \int_{t_0}^{t} f(\tau)\mathrm{d}\tau \Big|_{t_0}^{T}.$$

Proof. Let $F = \widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}$. Since F is bounded, there is a $K > 0$ such that $\left|F(P^t)\right| < K$ for all P^t.

Since S is finite, we can let $\{T_1, \cdots, T_k\} = S$ for some $k \in \mathbb{N}$.

Given $\epsilon > 0$, let $\gamma = \dfrac{\epsilon/2}{(3k+3)K' + T - t_0} > 0$, where $K' = K + \max\{|f(\tau)| \mid \tau \in [T_0, T]\}$.

1. For all $t \geq t_0$,

$$\left| \widetilde{\text{at}}[\![\textbf{integral } b]\!]_{\mathcal{B}}\ P^t - \int_{t_0}^{t} f(\tau)\mathrm{d}\tau \right|$$

$$\leq \quad e_1 + e_2 \text{ where } \begin{cases} P^t & \equiv & \langle t_1, \cdots, t_n \rangle \\ m & = & \min\{j \in \mathbb{N}_n \mid t_j \geq t_0\} \\ a & = & f(t_0) \cdot (t_m - t_0) + \Sigma_{j=m}^{n-1} f(t_j)\Delta t_j \\ e_1 & = & \left| \widetilde{\text{at}}[\![\textbf{integral } b]\!]_{\mathcal{B}}\ P^t - a \right| \\ e_2 & = & \left| a - \int_{t_0}^{t} f(\tau)\mathrm{d}\tau \right| \end{cases}$$

Since $F(P^t) \rightarrowtail f(t)\big|_{t_0}^{T} - S$, there is $\delta_1 \in (0, \gamma)$ such that $t \in [0,T] - [S]_\gamma$ and $\left|P^t\right|_{t_0} < \delta_1$ imply $\left| F(P^t) - f(t) \right| < \gamma$.

Therefore, when $t \in [t_0, T]$ and $\left|P^t\right|_{t_0} < \delta_1$, letting $M = \{j \in \{m..n-1\} \mid t_j \in [S]_\gamma\}$,

we have

$$
\begin{aligned}
e_1 &= \left|\Sigma_{j=1}^{n-1} v_j \Delta t_j - a\right| \qquad \text{where } v_j = F\,\langle t_1, \cdots, t_j\rangle \\
&\qquad \text{(operational semantics of } \textbf{integral}\text{)} \\
&= \left|\Sigma_{j=1}^{m-1} v_j \Delta t_j + \Sigma_{j=m}^{n-1} v_j \Delta t_j - f(t_0)\cdot(t_m - t_0) - \Sigma_{j=m}^{n-1} f(t_j)\Delta t_j\right| \\
&= \left|\Sigma_{j=1}^{m-1} v_j \Delta t_j - f(t_0)\cdot(t_m - t_0) + \Sigma_{j=m}^{n-1} v_j \Delta t_j - \Sigma_{j=m}^{n-1} f(t_j)\Delta t_j\right| \\
&\leq \left|\Sigma_{j=1}^{m-1} v_j \Delta t_j\right| + |f(t_0)\cdot(t_m - t_0)| + \Sigma_{j=m}^{n-1} |v_j - f(t_j)|\Delta t_j \\
&< K'\cdot(t_m - t_1) + K'\cdot(t_m - t_0) + \Sigma_{j=m}^{n-1} |v_j - f(t_j)|\Delta t_j \\
&= K'\cdot((t_m - t_0) + (t_0 - t_1) + (t_m - t_0)) + \Sigma_{j=m}^{n-1} |v_j - f(t_j)|\Delta t_j \\
&< 3K'\delta_1 + \Sigma_{j=m}^{n-1} |v_j - f(t_j)|\Delta t_j \\
&\qquad \left(\left|P^t\right|_{t_0} < \delta_1\right) \\
&< 3K'\delta_1 + \Sigma_{j\in\{m..n-1\}-M} |v_j - f(t_j)|\Delta t_j + \Sigma_{j\in M} |v_j - f(t_j)|\Delta t_j \\
&< 3K'\delta_1 + \Sigma_{j\in\{m..n-1\}-M} (\gamma \Delta t_j) + \Sigma_{j\in M} |v_j - f(t_j)|\Delta t_j \\
&\leq 3K'\delta_1 + (T - t_0)\gamma + \Sigma_{j\in M} |v_j - f(t_j)|\Delta t_j \\
&< 3K'\delta_1 + (T - t_0)\gamma + \Sigma_{j\in M} K'\Delta t_j \\
&< 3K'\delta_1 + (T - t_0)\gamma + K'\cdot k\cdot(2\gamma + \delta_1) \\
&\qquad \left(\left|P^t\right|_{t_0} < \delta_1 \text{ and by the definition of } M\right) \\
&< 3K'\gamma + (T - t_0)\gamma + 3kK'\gamma \\
&\qquad (\delta_1 < \gamma) \\
&= ((3k + 3)K' + T - t_0)\gamma \\
&= \epsilon/2
\end{aligned}
$$

By Lemma A.3, we know there is $\delta_2 > 0$ (depending only on ϵ), such that for all $t \in [t_0, T]$ and any non-empty non-descending sequence $Q = \langle t_0, \cdots, t\rangle$,

$$
|Q| < \delta_2 \implies \left|\Sigma_{j=1}^{n-1} f(\tau_j)\Delta\tau_j - \int_{t_0}^{t} f(\tau)\mathrm{d}\tau\right| \leq \epsilon/2
$$

where $Q \equiv \langle \tau_1, \cdots, \tau_n\rangle$.

Since $\left|P^t\right|_{t_0} < \delta_2 \implies |\langle t_0, t_m, t_{m+1}, \cdots, t_n\rangle| < \delta_2$, we have that

$$e_2 \leq \epsilon/2.$$

Let $\delta = \min\{\delta_1, \delta_2\}$. We have that for all $t \in [t_0, T]$, and any non-empty non-descending sequence P^t,

$$\left|P^t\right|_{t_0} < \delta \implies \left| \widetilde{\text{at}}[\![\textbf{integral } b]\!]_{\mathcal{B}}\ P^t - \int_{t_0}^{t} f(\tau)\mathrm{d}\tau \right| \leq e_1 + e_2 < \epsilon.$$

2. When $t < t_0$, for P^t where $\left|P^t\right|_{t_0} < \delta$, we have

$$\begin{array}{rl} & \left| Tat\textbf{integral } b_{\mathcal{B}}\ P^t - \int_{t_0}^{t}\ f(\tau)\mathrm{d}\tau \right| \\ = & \left| Tat\textbf{integral } b_{\mathcal{B}}\ P^t + \int_{t_0}^{t}\ f(\tau)\mathrm{d}\tau \right| \\ \leq & \left| Tat\textbf{integral } b_{\mathcal{B}}\ P^t \right| + \left| \int_{t_0}^{t}\ f(\tau)\mathrm{d}\tau \right| \\ < & (t - t_1)K' + (t_0 - t)K' \\ = & (t_0 - t_1)K' \\ < & K'\delta \\ < & K'\delta_1 \\ < & K'\gamma \\ < & \epsilon \end{array}$$

Since ϵ is arbitrary, we have proved that $\widetilde{\text{at}}[\![\textbf{integral } b]\!]_{\mathcal{B}}\ P^t \rightarrowtail \int_{t_0}^{t} f(\tau)\mathrm{d}\tau\Big|_{t_0}^{T}$. □

Theorem 4.6 (Let-behavior). Given $b \equiv$ **let behavior** $\{z_j(x_j) = w_j\}$ **in** b', we have that

$$\vdash b;\ \mathcal{B} \text{ and } \widetilde{\text{at}}[\![b']\!]_{\mathcal{B}\cup\{z_j \mapsto \lambda x_j.w_j\}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T} - S$$

imply that

$$\widetilde{\text{at}}[\![b']\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T} - S.$$

Proof. By the operational semantics for **let-behavior**, we know that

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t = \widetilde{\text{at}}[\![b']\!]_{\mathcal{B}\cup\{z_j \mapsto \lambda x_j.w_j\}}\ P^t,$$

and we are done. □

Theorem 4.7 (Until). Given $b \equiv b'$ **until** $\langle ev_j \Rightarrow z_j \rangle^{j \in \mathbb{N}_n}$, and $\vdash b$; $\mathcal{B}$,

1. if $\forall j \in \mathbb{N}_n.\ \widetilde{\text{occ}}^*[\![ev_j]\!]_{\mathcal{B}}\ t_0\ T = \textsf{Nothing}$ and $\widetilde{\text{at}}[\![b']\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\big|_{t_0}^{T} - S$, then

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\big|_{t_0}^{T} - S;$$

2. if $\exists T' \in (t_0, T],\ k \in \mathbb{N}_n$, such that

   (a) $\widetilde{\text{occ}}^*[\![ev_k]\!]_{\mathcal{B}}\ t_0\ T' = \textsf{Just}\ (\tau, v)$,

   (b) $\forall j \in \mathbb{N}_n - \{k\}.\ \widetilde{\text{occ}}^*[\![ev_j]\!]_{\mathcal{B}}\ t_0\ T' = \textsf{Nothing}$,

   (c) $\widetilde{\text{at}}[\![b']\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\big|_{t_0}^{\tau} - S$,

   (d) $\widetilde{\text{at}}[\![w[x := u]]\!]_{\mathcal{B}}\ P^t \rightarrowtail g(u,t)\big|_{\tau}^{T} - S'$, where $\lambda x.w \equiv \mathcal{B}(z_k)$, and

   (e) $\forall \epsilon > 0.\ \exists \delta > 0.\ \forall t \in [\tau, T].\ |\, u - v \,| < \delta \implies |\, g(u,t) - g(v,t) \,| < \epsilon$,

   then

$$\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail h(t)\big|_{\tau}^{T} - (S \cup S' \cup \{\tau\})$$

   where $h(t) =$ if $t < \tau$ then $f(t)$ else $g(v,t)$.

Proof. Let $F = \widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}$, $F' = \widetilde{\text{at}}[\![b']\!]_{\mathcal{B}}$, and $F_j = \widetilde{\text{occ}}[\![ev_j]\!]_{\mathcal{B}}$.

1. The first part of the theorem is proved as follows:

   Since $\widetilde{\text{occ}}^*[\![ev_j]\!]_{\mathcal{B}}\ t_0\ T = \textsf{Nothing}$, we know that given $\epsilon > 0$, for all $j \in \mathbb{N}_n$, there is $\delta_j > 0$ such that

$$\forall t \in [t_0, T].\ \left|P^t\right|_{t_0} < \delta_j \implies F_j(P^t) = \textsf{Nothing}$$

(by the definition of $\widetilde{\text{occ}}[\![-]\!]$ and $\widetilde{\text{occ}}^*[\![-]\!]$).

By the operational semantics of **until**, for all $t \in [0,T]$ and P^t where $\left|P^t\right|_{t_0} < \delta$ (we define δ as $\min\{\delta_j\}^{j\in\mathbb{N}_n}$), we have

$$F(P^t) = F'(P^t).$$

And we are done.

2. Next we prove the second part of the theorem.

   Given any $\epsilon > 0$, for all $t \in [0,T] - [S \cup S' \cup \{\tau\}]_\epsilon$, there are two possibilities:

   (a) $t \in [0,\tau] - [S \cup S' \cup \{\tau\}]_\epsilon$.

   There is $\delta_1 > 0$, such that for all P^t where $\left|P^t\right|_{t_0} < \delta_1$, we have

   $$\forall j \in \mathbb{N}_n.\ F_j(P^t) = \mathsf{Nothing}.$$

   Since $F'(P^t) \rightarrowtail f(t)\big|_{t_0}^{\tau} - S$, we can choose a δ_1 which makes $\left|\, F'(P^t) - f(t) \,\right| < \epsilon$.

   By the operational semantics of **until**, we know that

   $$\left|\, F(P^t) - h(t) \,\right| = \left|\, F'(P^t) - h(t) \,\right| = \left|\, F'(P^t) - f(t) \,\right| < \epsilon.$$

   (b) Otherwise $t \in [\tau,T] - [S \cup S' \cup \{\tau\}]_\epsilon$.

   Since $\forall \epsilon > 0.\ \exists \delta > 0.\ \forall t \in [\tau,T].\ |\, u - v \,| < \delta \implies |\, g(u,t) - g(v,t) \,| < \epsilon$, we know there is $\gamma > 0$ (depending only on ϵ) such that

   $$|\, u - v \,| < \gamma \implies |\, g(u,t) - g(v,t) \,| < \epsilon/2.$$

   Since $\widetilde{\text{at}}[\![w[x := u]]\!]_{\mathcal{B}}\ P^t \rightarrowtail g(u,t)\big|_{\tau}^{T} - S'$, we know that there is $\delta_2 > 0$ (depend-

ing only on ϵ), such that

$$\left|Q^t\right|_\tau < \delta_2 \implies \left| G(u, Q^t) - g(u,t) \right| < \epsilon/2$$

where $G(u, Q^t) \stackrel{\text{def}}{=} \widetilde{\text{at}}[\![w[x := u]]\!]_{\mathcal{B}}\ Q^t$.

Now, we pick an arbitrary $\epsilon_3 \in (0, \min\{\gamma, \delta_2/2\})$. By the definition of $\widetilde{\text{occ}}^*[\![-]\!]$, we know that there is $\delta_3 \in (0, \delta_2/2)$ (depending only on ϵ_3), τ', and u such that

$$\left|P^t\right|_{t_0} < \delta_3 \implies F_k(P^t) = \textsf{Just}\ (\tau', u) \text{ and } \forall j \in \mathbb{N}_n - \{k\}.\ F_j(R) = \textsf{Nothing}$$

where $|\tau' - \tau| < \epsilon_3$ and $|u - v| < \epsilon_3$, and $R = \langle t_1, \cdots, t_{m+1}\rangle$ where $P^t \equiv \langle t_1, \cdots, t_m = \tau', t_{m+1}, \cdots, t_n\rangle$.

let $Q^t = \langle t_{m+1}, \cdots, t_n\rangle$ where $P^t \equiv \langle t_1, \cdots, t_m = \tau', t_{m+1}, \cdots, t_n\rangle$.

By the operational semantics of **until**, we have that

$$F(P^t) = G(u, Q^t).$$

Since $\left|Q^t\right|_\tau < \delta_3 + \epsilon_3 < \delta_2/2 + \delta_2/2 = \delta_2$, we know that

$$\left| G(u, Q^t) - g(u,t) \right| < \epsilon/2.$$

Therefore,

$$\begin{aligned}
\left| F(P^t) - f(t) \right| &= \left| F(P^t) - g(v,t) \right| \\
&= \left| G(u, Q^t) - g(v,t) \right| \\
&= \left| G(u, Q^t) - g(u,t) + g(u,t) - g(v,t) \right| \\
&\leq \left| G(u, Q^t) - g(u,t) \right| + \left| g(u,t) - g(v,t) \right| \\
&< \epsilon/2 + \epsilon/2 \\
&= \epsilon.
\end{aligned}$$

Combining the above two cases, we know that for any given $\epsilon > 0$, there is $\delta = \min\{\delta_1, \delta_3\} > 0$, such that for all $t \in [t_0, T] - [S \cup S' \cup \{\tau\}]_\epsilon$ and P^t where $\left|P^t\right|_{t_0} < \delta$, we have $\left| F(P^t) - h(t) \right| < \epsilon$. And we are done.

□

Theorem 4.8 (When). Given $ev \equiv$ **when** b and $\vdash ev;\ \mathcal{B}$, we have

1. if $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T}$, and there are T_0, T_1 where $T_0 \lessdot t_0 \leq T_1 \leq T$, such that

   (a) $T_0 \leq t < T_1 \implies f(t) =$ **True**, and

   (b) $T_1 < t < T \implies f(t) =$ **False**,

   then $\widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T =$ Nothing;

2. if there are $T_0, T_1, T_2, T_3 \in \mathbb{T}$ where $T_0 \lessdot t_0 \leq T_1 < T_2 < T_3 \leq T$, such that

   (a) $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t \rightarrowtail f(t)\Big|_{t_0}^{T_3}$,

   (b) $t \in [T_0, T_1) \cup (T_2, T_3] \implies f(t) =$ **True**, and

   (c) $t \in (T_1, T_2) \implies f(t) =$ **False**,

   then $\widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T =$ Just $(T_2, ())$.

Proof. 1. First we prove part 1 of the theorem. We can find a $\delta > 0$ such that for all $t \in [0, T]$ and P^t where $\left|P^t\right|_{t_0} < \delta$, we have $\left| \widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t - f(t) \right| < 0.5$. Since b is a Boolean behavior, whose only two possible values are True and False, where $|\,\text{True} - \text{False}\,| = 1$, this practically means that $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t = f(t)$.

Let P^T be a time sequence where $\left|P^T\right|_{t_0} < \delta$. Assume $P^T = \langle t_1, \cdots, t_n \rangle$ and $\widetilde{\text{trace}}[\![b]\!]_{\mathcal{B}}\ P^T = \langle v_1, \cdots, v_n \rangle$. We have that there is $k \in \mathbb{N}_{n-1}$ where $\forall j < k.\ v_j =$ True, and $\forall j \in \{k+1..n-1\}.\ v_j =$ False. Thus, by the definition of $\widetilde{\text{occ}}[\![-]\!]$, we know $\widetilde{\text{occ}}[\![ev]\!]_{\mathcal{B}}\ P^T =$ Nothing. Since P^T is arbitrary, we have that $\widetilde{\text{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T =$ Nothing.

2. The proof for the second part is similar. Given an $\epsilon > 0$, we can choose a $\delta > 0$ such that

   (a) $\delta < \dfrac{\min\{\epsilon, T_2 - T_1, T_3 - T_2\}}{2}$,

   (b) if $t_0 > T_0$, then $\delta < t_0 - T_0$, and

   (c) for all $t \in [0, T_3]$ and P^t where $\left|P^t\right|_{t_0} < \delta$, we have $\widetilde{\text{at}}[\![b]\!]_{\mathcal{B}}\ P^t = f(t)$.

   Let P^T be a time sequence where $\left|P^T\right|_{t_0} < \delta$. Assume $P^T = \langle t_1, \cdots, t_n \rangle$ and $\widetilde{\text{trace}}[\![b]\!]_{\mathcal{B}}\ P^T = \langle v_1, \cdots, v_n \rangle$. It is obvious that $\forall j \in \mathbb{N}_n .\ v_j = f(t_j)$.

   We proceed by comparing the first element in P^T (i.e. t_1) with T_1:

   (a) if $t_1 \geq T_1$, then for all $j \in \mathbb{N}_n .\ t_j \notin [T_0, T_1)$;

   (b) otherwise $t_1 < T_1$. Since $T_0 \lessdot t_0$, there are in turn two possibilities:

      i. $T_0 = t_0 = 0$. We know that $t_1 \geq T_0$, since $t_1 \in \mathbb{T}$.

      ii. $T_0 < t_0$. Since $|\, t_1 - t_0 \,| < \delta$ and $\delta < t_0 - T_0$, we also know that $t_1 \geq T_0$.

      In either case, there is $m \in \mathbb{N}_n .\ j \in \mathbb{N}_m \iff t_j \in [T_0, T_1)$.

   Therefore, no matter t_1 is less than T_1 or not, there is $m \in \{0..n\}$ such that $\forall j \in \mathbb{N}_n .\ j \in \mathbb{N}_m \iff t_j \in [T_0, T_1)$.

   Since $\left|P^T\right|_{t_0} < \delta$, $\delta < \dfrac{T_2 - T_1}{2}$, and $\delta < \dfrac{T_3 - T_2}{2}$. we know that there is $k \in \{m+2..n\}$ such that $t_{m+2}, t_{m+3}, \cdots, t_k \in (T_1, T_2)$ and $t_{k+2} \in (T_2, T_3]$.

   Since $\forall j \in \mathbb{N}_n .\ v_j = f(t_j)$, we now have that

   (a) for all $j \leq m$, $v_j = \mathsf{True}$,

   (b) $v_{m+2} = v_{m+3} = \cdots = v_k = \mathsf{False}$, where $k \geq m + 2$, and

   (c) $v_{k+2} = \mathsf{True}$.

   Let $k' = k + 1$ (if $v_{k+1} = \mathsf{True}$) or $k + 2$ (otherwise). By the operational semantics of **when**, we have $\widetilde{\text{occ}}[\![ev]\!]_{\mathcal{B}}\ P^T = \mathsf{Just}\ (t_{k'}, ())$.

Since $|\, t_{k'} - T_2 \,| < 2\delta < \epsilon$, and ϵ is arbitrary, we know that $\widetilde{\mathsf{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T = \mathsf{Just}\ (T_2, ())$.

Now we have proved both parts of the theorem, and we are done. □

Theorem 4.9 (Liftev). Given $ev \equiv$ **liftev** $(\lambda(x_0, \cdots, x_n).e)\ ev'\ b_1\ \cdots\ b_n$ and $\mathcal{B}$ where $\vdash ev;\ \mathcal{B}$, write oc for $\widetilde{\mathsf{occ}}^*[\![ev]\!]_{\mathcal{B}}\ t_0\ T$ and oc' for $\widetilde{\mathsf{occ}}^*[\![ev']\!]_{\mathcal{B}}\ t_0\ T$. We have

1. $oc' = \mathsf{Nothing} \implies oc = \mathsf{Nothing}$, and
2. if
   (a) $oc' = \mathsf{Just}\ (\tau, v_0)$,
   (b) there is $\xi > 0$ such that for all $j \in \mathbb{N}_n$,
      i. $\widetilde{\mathsf{at}}[\![b_j]\!]_{\mathcal{B}}\ P^t \rightarrowtail f_j(t)\big|_{t_0}^{T} - S_j$ where $\tau \notin [S_j]_\xi$, and
      ii. $f_j(t)$ is continuous at $t = \tau$,

      and
   (c) the function $g(y_0, y_1, \cdots, y_n) = \mathbf{eval}[\![e]\!]_{\{x_j \mapsto y_j\}^{j \in \{0..n\}}}$ is continuous at $(v_0, v_1, \cdots, v_n)$, where $\forall j \in \mathbb{N}_n.\ v_j = f_j(\tau)$,

   then $oc = \mathsf{Just}\ (\tau, g(v_0, v_1, \cdots, v_n))$.

Proof. 1. By the operational semantics of **liftev**, $\widetilde{\mathsf{occ}}[\![ev']\!]_{\mathcal{B}}\ P^T = \mathsf{Nothing} \implies \widetilde{\mathsf{occ}}[\![ev]\!]_{\mathcal{B}}\ P^T = \mathsf{Nothing}$. Since $oc' = \lim_{|P^T|_{t_0} \to 0} \widetilde{\mathsf{occ}}[\![ev']\!]_{\mathcal{B}}\ P^T$, and $oc = \lim_{|P^T|_{t_0} \to 0} \widetilde{\mathsf{occ}}[\![ev]\!]_{\mathcal{B}}\ P^T$, we know that $oc' = \mathsf{Nothing} \implies oc = \mathsf{Nothing}$

2. Given any $\epsilon > 0$, since g is continuous at $(v_0, v_1, \cdots, v_n)$, there is $\gamma_1 > 0$ such that

$$|\, (v_0', v_1', \cdots, v_n') - (v_0, v_1, \cdots, v_n) \,| < \gamma_1 \implies$$
$$|\, g(v_0', v_1', \cdots, v_n') - g(v_0, v_1, \cdots, v_n) \,| < \epsilon/2.$$

Since $\forall j \in \mathbb{N}_n.\ f_j(t)$ is continuous at $t = \tau$, there is $\gamma_2 > 0$ such that $|\, \tau' - \tau \,| < \gamma_2 \implies \forall j \in \mathbb{N}_n. \left| v_j' - v_j \right| < \dfrac{\gamma_1}{2n+1}$ where $v_j' = f_j(\tau')$.

Let $\gamma_3 = \min\{\epsilon/2, \xi/2, \dfrac{\gamma_1}{2n+1}, \gamma_2\}$. Since $oc' = \mathsf{Just}\ (\tau, v_0)$, there is $\delta_1 > 0$ (depending only on γ_3) such that $\left|P^T\right|_{t_0} < \delta_1 \implies \widetilde{\mathsf{occ}}[\![ev']\!]_{\mathcal{B}}\ P^T = \mathsf{Just}\ (\tau', v_0')$ where $|\ \tau' - \tau\ | < \gamma_3$ and $|\ v_0' - v_0\ | < \gamma_3$.

Since $|\ \tau' - \tau\ | < \gamma_3$, $\tau \notin [S_j]_{\xi}$, and $\gamma_3 + \gamma_3 \leq \xi$, we know that $\tau' \notin [S_j]_{\gamma_3}$ for all $j \in \mathbb{N}_n$.

Since $\widetilde{\mathsf{at}}[\![b_j]\!]_{\mathcal{B}}\ P^t \rightarrowtail f_j(t)\Big|_{t_0}^{T} - S_j$ and $\tau' \notin [S_j]_{\gamma_3}$, there is $\delta_2 > 0$ (depending only on γ_3) such that $\left|P^{\tau'}\right|_{t_0} < \delta_2$ implies $\left|\ \widetilde{\mathsf{at}}[\![b_j]\!]_{\mathcal{B}}\ P^{\tau'} - f_j(\tau')\ \right| < \gamma_3$.

Therefore, for any P^T where $\left|P^T\right|_{t_0} < \delta = \min\{\delta_1, \delta_2\}$, we have

$$
\begin{aligned}
& \left|\ \widetilde{\mathsf{occ}}[\![ev]\!]_{\mathcal{B}}\ P^T - \mathsf{Just}\ (\tau, g(v_0, v_1, \cdots, v_n))\ \right| \\
= \quad & |\ \mathsf{Just}\ (\tau', g(v_0', u_1, \cdots, u_n)) - \mathsf{Just}\ (\tau, g(v_0, v_1, \cdots, v_n))\ | \\
& \qquad \text{where } \mathsf{Just}\ (\tau', v_0') = \widetilde{\mathsf{occ}}[\![ev']\!]_{\mathcal{B}}\ P^T,\ u_j = \widetilde{\mathsf{at}}[\![b_j]\!]_{\mathcal{B}}\ Q^{\tau'}, \text{ and} \\
& \qquad\qquad Q^{\tau'} \text{ is the prefix of } P^T \text{ ending at } \tau' \\
& \qquad\qquad \text{(by the operational semantics of } \textbf{liftev}\text{)} \\
= \quad & |\ \tau' - \tau\ | + |\ g(v_0', u_1, \cdots, u_n) - g(v_0, v_1, \cdots, v_n)\ |
\end{aligned}
$$

Since $\left|P^T\right|_{t_0} < \delta \leq \delta_1$, we know $|\ \tau' - \tau\ | < \gamma_3$ and $|\ v_0' - v_0\ | < \gamma_3$.

Since $\left|P^T\right|_{t_0} < \delta \leq \delta_2$, we know $\forall j \in \mathbb{N}_n .\ \left|\ u_j - v_j'\ \right| < \gamma_3$.

Since $|\ \tau' - \tau\ | < \gamma_3 \leq \gamma_2$, we also know that $\forall j \in \mathbb{N}_n .\ \left|\ v_j' - v_j\ \right| < \dfrac{\gamma_1}{2n+1}$.

Hence,

$$\begin{aligned}
& |\,(v_0', u_1, \cdots, u_n) - (v_0, v_1, \cdots, v_n)\,| \\
= \;& |\,v_0' - v_0\,| + \Sigma_{j=1}^{n} |\,u_j - v_j\,| \\
< \;& \gamma_3 + \Sigma_{j=1}^{n} |\,u_j - v_j\,| \\
= \;& \gamma_3 + \Sigma_{j=1}^{n} \left| u_j - v_j' + v_j' - v_j \right| \\
\leq \;& \gamma_3 + \Sigma_{j=1}^{n} \left(\left| u_j - v_j' \right| + \left| v_j' - v_j \right| \right) \\
< \;& \gamma_3 + \Sigma_{j=1}^{n} \left(\gamma_3 + \frac{\gamma_1}{2n+1} \right) \\
\leq \;& \gamma_3 + \Sigma_{j=1}^{n} \left(\frac{\gamma_1}{2n+1} + \frac{\gamma_1}{2n+1} \right) \\
= \;& \gamma_3 + \frac{2n\gamma_1}{2n+1} \\
\leq \;& \frac{\gamma_1}{2n+1} + \frac{2n\gamma_1}{2n+1} \\
= \;& \gamma_1,
\end{aligned}$$

which in turn leads to

$$\left|\, g(v_0', v_1', \cdots, v_n') - g(v_0, v_1, \cdots, v_n) \,\right| < \epsilon/2.$$

Therefore, now we have

$$\begin{aligned}
& \left|\, \widetilde{\mathsf{occ}}[\![ev]\!]_{\mathcal{B}}\, P^T - \mathsf{Just}\,(\tau, g(v_0, v_1, \cdots, v_n)) \,\right| \\
= \;& |\,\tau' - \tau\,| + |\,g(v_0', u_1, \cdots, u_n) - g(v_0, v_1, \cdots, v_n)\,| \\
< \;& \gamma_3 + \epsilon/2 \\
\leq \;& \epsilon/2 + \epsilon/2 \\
= \;& \epsilon.
\end{aligned}$$

Since ϵ is arbitrary, we have proved that $oc = \mathsf{Just}\,(\tau, g(v_0, v_1, \cdots, v_n))$.

□

A.3 Theorems in Chapter 5

Lemma 5.1 (Deterministic semantics). If the base language is deterministic, then we have that

$$(P \xrightarrow{I} \mathcal{X}; P \text{ and } P \xrightarrow{I} \mathcal{X}'; P') \implies (\mathcal{X} \equiv \mathcal{X}' \text{ and } P \equiv P').$$

Proof. We prove the lemma in two steps.

1. First, we prove that $P \vdash d \xrightarrow{I} v$ and $P \vdash d \xrightarrow{I} v'$ imply that $v \equiv v'$.

   The proof is by induction on the derivation trees for $P \vdash d \xrightarrow{I} v$. Let's call this derivation $D1$ and that for and $P \vdash d \xrightarrow{I} v'$ $D2$. There are three cases:

   (a) The last rule used in $D1$ is e1. The term d must have the form e (a base language term). Hence the last rule in $D2$ must also be e1. By induction hypothesis and the deterministic semantics of the base language, we have that $v \equiv v'$.

   (b) The last rule used in $D1$ is e2. By similar reasoning as above, the last rule in $D2$ must be e2 too. Then we get by the deterministic semantics of the base language that $v \equiv v'$.

   (c) The last rule in $D1$ is e3. In this case, the last rule in $D2$ must also be e3. It then follows trivially that $v \equiv v'$.

2. Next, we prove that $P \vdash d \xrightarrow{I} d'$ and $P \vdash d \xrightarrow{I} d''$ imply that $d' \equiv d''$.

   Let $D1$ be the derivation tree for $P \vdash d \xrightarrow{I} d'$, and $D2$ be that for $P \vdash d \xrightarrow{I} d''$. The proof is by induction on $D1$.

   (a) The last rule used in $D1$ is u1. We have $d \equiv e \equiv d'$ and $d \equiv e \equiv d''$. Therefore, $d' \equiv d''$.

   (b) The last rule in $D1$ is u2. The last rule in $D2$ has to be u2 too. By part one of the proof, we know that $d' \equiv d''$.

   (c) The last rule in $D1$ is u3. We have $d \equiv d'$ and $d \equiv d''$. Hence $d' \equiv d''$.

Combining part 1 and 2, by rule p, we know that

$$P \xrightarrow{I} \mathcal{X}; P \wedge P \xrightarrow{I} \mathcal{X}'; P' \implies \mathcal{X} \equiv \mathcal{X}' \wedge P \equiv P'.$$

□

Theorem 5.1 (Source-target correspondence). Given that $\Gamma \vdash P$, $P \rightsquigarrow Q$, and $P \xrightarrow{I} \mathcal{X}; P'$, we have

1. $P' \rightsquigarrow Q$; and
2. if $Q \vdash \mathrm{state}(P) \overset{I}{\hookrightarrow} \mathcal{X}'; \mathcal{X}''$, then $\mathcal{X}' \supseteq \mathcal{X}$ and $\mathcal{X}'' \equiv \mathrm{state}(P')$.

In the proof that follows, we use the notation

$$\mathcal{X}|X$$

for the environment obtained from restricting $\mathcal{X}$ to variables in X, i.e.

$$\{x \mapsto \mathcal{X}(x) \mid x \in X\}.$$

Proof. Based on the form of the behavior definitions, we can divide the source program P into four disjoint sets:

$$\begin{aligned} P \quad \equiv \quad & \{x_i = e_i\}^{i \in J} \uplus \\ & \{x_i = \textbf{init}\, y_i = v_i \textbf{ in } \{I \Rightarrow e_i\} \uplus H_i\}^{i \in K} \uplus \\ & \{x_i = \textbf{init}\, y_i = v_i \textbf{ in } \{I \Rightarrow e_i \textbf{ later}\} \uplus H_i\}^{i \in L} \uplus \\ & \{x_i = \textbf{init}\, y_i = v_i \textbf{ in } H_i\}^{i \in M} \end{aligned}$$

where J, K, L, and M are disjoint sets of natural numbers, and for all $i \in M$, H_i does not contain I.

By the definition of $P \rightsquigarrow Q$, we know that for any $I \in \mathcal{I}$, there are A and A' such that

$P \vdash_I^1 A$ and $P \vdash_I^2 A'$. The core of the proof is by induction on the derivation of $P \vdash_I^1 A$ and $P \vdash_I^2 A'$.

Part 1: By inspecting the operational semantics of E-FRP, we know that

$$\begin{aligned} P' \quad \equiv \quad & \{x_i = e_i\}^{i \in J} \uplus \\ & \{x_i = \textbf{init}\, y_i = v_i' \,\textbf{in}\, \{I \Rightarrow e_i\} \uplus H_i\}^{i \in K} \uplus \\ & \{x_i = \textbf{init}\, y_i = v_i' \,\textbf{in}\, \{I \Rightarrow e_i \,\textbf{later}\} \uplus H_i\}^{i \in L} \uplus \\ & \{x_i = \textbf{init}\, y_i = v_i \,\textbf{in}\, H_i\}^{i \in M} \end{aligned}$$

for some $\{v_i'\}^{i \in K \uplus L}$.

Given this, we can easily verify that $P' \vdash_I^1 A$ and $P' \vdash_I^2 A'$, and therefore $P' \rightsquigarrow Q$.

Part 2: From now on, we let $\mathcal{X}_0 = \text{state}(P)$, $\mathcal{X}_1 = \text{state}(P')$, $N = J \uplus K \uplus L \uplus M$, and $n =$ the size of N. We want to prove the following proposition:

Proposition 1. For any $i \in N$,

1. $\mathcal{X}'(x_i) = \mathcal{X}(x_i)$;
2. if $i \in L$, then $\mathcal{X}'(x_i^+) = \mathcal{X}_1(x_i)$; and
3. if $i \in M$, then $\mathcal{X}'(x_i^+) = \mathcal{X}'(x_i)$.

Since Proposition 1 is establishing a property of i, for brevity we give this property a name p and write $p(i)$ for that i has this property. Hence Proposition 1 can be expressed as "$\forall i \in N.\, p(i)$." To show that this is true, we note that the derivation of $P \vdash_I^1 A$ has the form:

$$\dfrac{\dfrac{\dfrac{}{P_n \vdash_I^1 A_n}\ \text{side-condition}_n}{P_{n-1} \vdash_I^1 A_{n-1}}\ \text{side-condition}_{n-1}}{\vdots}$$
$$P_0 \vdash_I^1 A_0$$

where $P_n \equiv \emptyset$, $A_n \equiv \langle\rangle$, $P_0 \equiv P$, $A_0 \equiv A$, and for all $j \in \mathbb{N}_n$, $P_{j-1} \equiv \{x_{i_j} = \cdots\} \uplus P_{j-1}$. It is clear that A assigns to any variable at most once.

Apparently, $\langle i_1, \cdots, i_n \rangle$ is a permutation of N. Therefore we can prove Proposition 1 by showing that the following equivalent proposition is true:

Proposition 2. For any $j \in \mathbb{N}_n$, we have $p(i_j)$.

The proof for Proposition 2 is by induction on j. Given $k \in \mathbb{N}_n$, assuming that $\forall j \in \mathbb{N}_{k-1}.\ p(i_j)$, we try to prove that $p(i_k)$:

Let $i = i_k$. There are four cases:

1. $i \in J$. We know that $(x_i = e_i) \in P$, and that the rule leading to $P_{k-1} \vdash_I^1 A_{k-1}$ is x2. Hence $A_{k-1} \equiv \langle x_i := e_i \rangle + A_k$ and $x_i := e_i < A_k$.

   Since $x_i := e_i < A_k$, we have that for all $x \in FV(e_i)$, it has to be in one of the two following cases:

   (a) $x \in \{x_{i_j}\}^{j \in \mathbb{N}_{k-1}}$. By the induction hypothesis, $\mathcal{X}'(x) = \mathcal{X}(x)$.

   (b) $x \equiv x_{i_j}$ for some $j \geq k$. Let $l = i_j$. This again can be divided into two cases:

      i. $(x_l = \textbf{init}\, y_l = v_l \textbf{ in } \{I \Rightarrow e_l \textbf{ later}\} \uplus H_l) \in P$. In this case, $\mathcal{X}'(x) = \mathcal{X}_0(x) = v_l = \mathcal{X}(x)$.

      ii. $(x_l = \textbf{init}\, y_l = v_l \textbf{ in } H_l) \in P$, where I does not occur in H_l. In this case, we also have $\mathcal{X}'(x) = \mathcal{X}_0(x) = v_l = \mathcal{X}(x)$.

   Therefore we have $\mathcal{X}'(x) = \mathcal{X}(x)$ for all $x \in FV(e_i)$.

   By rule c2, we have $\mathcal{X}'(x_i) = v'$ where $\mathcal{Y} \vdash e_i \hookrightarrow v'$ where $\mathcal{Y}(x) = \mathcal{X}'(x) = \mathcal{X}(x)$ for all $x \in FV(e_i)$. By rule e1, we know that $\mathcal{X}(x_i) = v$ where $\mathcal{X}|FV(e_i) \vdash e_i \hookrightarrow v$. Finally, by Lemma 2.3 we have that $v \equiv v'$, and therefore $\mathcal{X}'(x_i) = \mathcal{X}(x_i)$.

2. $i \in K$. We know that $(x_i = \textbf{init}\, y_i = v_i \textbf{ in } \{I \Rightarrow e_i\} \uplus H_i) \in P$ and the rule leading to $P_{k-1} \vdash_I^1 A_{k-1}$ is x3. Hence $A_{k-1} \equiv \langle x_i := e \rangle + A_k$ and $x_i := e < A_k$, where $e \equiv e_i[y_i := x_i^+]$. Since $x_i := e < A_k$, we know that for all $x \in FV(e)$, it must fall into one of the following three cases:

   (a) $x \equiv x_i^+$. $\mathcal{X}'(x) = \mathcal{X}_0(x_i^+) = v_i$.

(b) $x \in \{x_{i_j}\}^{j \in \mathbb{N}_{k-1}}$. By the induction hypothesis, $\mathcal{X}'(x) = \mathcal{X}(x)$.

(c) $x \equiv x_{i_j}$ for some $j \geq k$. Let $l = i_j$. By the same reasoning we used in bullet 1, we have that $\mathcal{X}'(x) = \mathcal{X}_0(x) = v_l = \mathcal{X}(x)$.

By rule c2, we have that $\mathcal{X}'(x_i) = v'$ where $\mathcal{Y} \vdash e \hookrightarrow v'$ where $\mathcal{Y}(x_i^+) = v_i$ and $\mathcal{Y}(x) = \mathcal{X}'(x) = \mathcal{X}(x)$ for all $x \in FV(e_i) - \{x_i^+\}$. By rule e2, we know that $\mathcal{X}(x_i) = v$ where $\mathcal{X}|FV(e_i[y_i := v_i]) \vdash e_i[y_i := v_i] \hookrightarrow v$. Thus by Lemma 2.8, we get that $v' = v$ and therefore $\mathcal{X}'(x_i) = \mathcal{X}(x_i)$.

3. $i \in L$. We know that $(x_i = \textbf{init}\, y_i = v_i \,\textbf{in}\, \{I \Rightarrow e_i \,\textbf{later}\} \uplus H_i) \in P$ and the rule leading to $P_{k-1} \vdash_I^1 A_{k-1}$ is x4. Hence $A_{k-1} \equiv \langle x_i^+ := e \rangle +\!\!+ A_k$ and $x_i^+ := e < A_k$, where $e \equiv e_i[y_i := x_i]$. Since A does not assign to x_i, we have that $\mathcal{X}'(x_i) = \mathcal{X}_0(x_i) = v_i = \mathcal{X}(x_i)$.

   Since $x_i^+ := e < A_k$, we have that for all $x \in FV(e)$, it has to be in one of the two following cases:

   (a) $x \in \{x_{i_j}\}^{j \in \mathbb{N}_{k-1}}$. By the induction hypothesis, $\mathcal{X}'(x) = \mathcal{X}(x)$.

   (b) $x \equiv x_{i_j}$ for some $j \geq k$. Let $l = i_j$. By the same reasoning we used in bullet 1, we have that $\mathcal{X}'(x) = \mathcal{X}_0(x) = v_l = \mathcal{X}(x)$.

   Therefore we have $\mathcal{X}'(x) = \mathcal{X}(x)$ for all $x \in FV(e)$.

   Since $A_{k-1} \equiv \langle x_i^+ := e \rangle +\!\!+ A_k$, we have that $\mathcal{X}'(x_i^+) = v'$ where $\mathcal{Y} \vdash e \hookrightarrow v'$ where $\mathcal{Y}(x) = \mathcal{X}'(x) = \mathcal{X}(x)$ for all $x \in FV(e)$.

   Meanwhile, by rule u2 we have that $\mathcal{X}_1(x_i) = v$ where $\mathcal{X} \vdash e_i[y_i := v_i] \hookrightarrow v$.

   Hence, by Lemma 2.8 we get $v' \equiv v$, and thus $\mathcal{X}'(x_i^+) = \mathcal{X}_1(x_i)$.

4. Otherwise $i \in M$. We know that $(x_i = \textbf{init}\, y_i = v_i \,\textbf{in}\, H_i) \in P$ where I does not occur in H_i, and the rule leading to $P_{k-1} \vdash_I^1 A_{k-1}$ is x5. Hence $A_{k-1} \equiv A_k$, and

therefore A does not assign to either x_i or x_i^+. Therefore we have

$$\mathcal{X}'(x_i) = \mathcal{X}_0(x_i) = v_i = \mathcal{X}(x_i),$$

and

$$\mathcal{X}'(x_i^+) = \mathcal{X}_0(x_i^+) = v_i = \mathcal{X}'(x_i).$$

Putting the above cases together, we have proved Proposition 2.

Next, we want to prove:

Proposition 3. For any $i \in N$, i has the property that

1. $\mathcal{X}''(x_i) = \mathcal{X}_1(x_i)$; and
2. if $i \in K \uplus L \uplus M$, then $\mathcal{X}''(x_i^+) = \mathcal{X}_1(x_i)$.

We will call this property q.

Note that the derivation of $P \vdash_I^2 A'$ has the form:

$$\dfrac{\dfrac{\dfrac{}{P_n \vdash_I^2 A_n}\ \text{side-condition}_n}{P_{n-1} \vdash_I^2 A_{n-1}}\ \text{side-condition}_{n-1}}{\vdots}$$
$$P_0 \vdash_I^2 A_0$$

where $P_n \equiv \emptyset$, $A_n \equiv \langle\rangle$, $P_0 \equiv P$, $A_0 \equiv A'$, and for all $j \in \mathbb{N}_n$, $P_{j-1} \equiv \{x_{i_j} = \cdots\} \uplus P_{j-1}$. It is clear that A' assigns to any variable at most once.

Apparently, $\langle i_1, \cdots, i_n \rangle$ is a permutation of N. Therefore we can show Proposition 1 is true by proving the following proposition, which is equivalent to Proposition 3:

Proposition 4. For any $j \in \mathbb{N}_n$, we have $q(i_j)$.

The proof is by induction on j: given $k \in \mathbb{N}_n$, assuming that $\forall j \in \mathbb{N}_{k-1}.\ q(i_j)$, we try to prove that $q(i_k)$:

Let $i = i_k$. There are four cases:

1. $i \in J$. We know that $(x_i = e_i) \in P$, and that the rule leading to $P_{k-1} \vdash_I^2 A_{k-1}$ is x7. Hence $A_{k-1} \equiv \langle x_i := e_i \rangle +\!\!+ A_k$ and $x_i := e_i < A_k$.

   Since $x_i := e_i < A_k$, we have that for all $x \in FV(e_i)$, it has to be in one of the two following cases:

   (a) $x \in \{x_{i_j}\}^{j \in \mathbb{N}_{k-1}}$. By the induction hypothesis, $\mathcal{X}''(x) = \mathcal{X}_1(x)$.

   (b) $x \equiv x_{i_j}$ for some $j \geq k$. Let $l = i_j$. This again can be divided into two cases:

      i. $(x_l = \textbf{init}\, y_l = v_l \textbf{ in } \{I \Rightarrow e_l\} \uplus H_l) \in P$. In this case, by Proposition 1 we have $\mathcal{X}''(x) = \mathcal{X}'(x) = \mathcal{X}(x) = \mathcal{X}_1(x)$.

      ii. $(x_l = \textbf{init}\, y_l = v_l \textbf{ in } H_l) \in P$, where I does not occur in H_l. In this case, we also have (by Proposition 1 again) that $\mathcal{X}''(x) = \mathcal{X}'(x) = \mathcal{X}(x) = \mathcal{X}_1(x)$.

   Therefore we have $\mathcal{X}''(x) = \mathcal{X}_1(x)$ for all $x \in FV(e_i)$.

   By rule c2, we know that $\mathcal{X}''(x_i) = v'$ where $\mathcal{Y} \vdash e_i \hookrightarrow v'$ where $\mathcal{Y}(x) = \mathcal{X}_1(x)$ for all $x \in FV(e_i)$. Meanwhile, by rule u1 we have $\mathcal{X}_1(x_i) = v$ where $\mathcal{X}_1 | FV(e_i) \vdash e_i \hookrightarrow v$. Therefore, by Lemma 2.3 we have $v' \equiv v'$, or $\mathcal{X}''(x_i) = \mathcal{X}_1(x_i)$.

2. $i \in K$. We know that $(x_i = \textbf{init}\, y_i = v_i \textbf{ in } \{I \Rightarrow e_i\} \uplus H_i) \in P$ and the rule leading to $P_{k-1} \vdash_I^1 A_{k-1}$ is x8. Hence $A_{k-1} \equiv \langle x_i^+ := x_i \rangle +\!\!+ A_k$ and A' does not assign to x_i. Therefore by Proposition 1, we have $\mathcal{X}''(x_i) = \mathcal{X}'(x_i) = \mathcal{X}(x_i) = \mathcal{X}_1(x_i)$ and $\mathcal{X}''(x_i^+) = \mathcal{X}'(x_i) = \mathcal{X}_1(x_i)$.

3. $i \in L$. We know that $(x_i = \textbf{init}\, y_i = v_i \textbf{ in } \{I \Rightarrow e_i \textbf{ later}\} \uplus H_i) \in P$ and the rule leading to $P_{k-1} \vdash_I^1 A_{k-1}$ is x9. Hence $A_{k-1} \equiv \langle x_i := x_i^+ \rangle +\!\!+ A_k$ and A' does not assign to x_i^+. Therefore by Proposition 1, we have $\mathcal{X}''(x_i) = \mathcal{X}'(x_i^+) = \mathcal{X}_1(x_i)$ and $\mathcal{X}''(x_i^+) = \mathcal{X}'(x_i^+) = \mathcal{X}_1(x_i)$.

4. $i \in M$. We know that $(x_i = \textbf{init}\, y_i = v_i \textbf{ in } H_i) \in P$ where I does not occur in H_i, and the rule leading to $P_{k-1} \vdash_I^1 A_{k-1}$ is x10. Hence $A_{k-1} \equiv A_k$, and therefore A' does

not assign to either x_i or x_i^+. Therefore by proposition 1 we have $\mathcal{X}''(x_i) = \mathcal{X}'(x_i) = \mathcal{X}(x_i) = v_i = \mathcal{X}_1(x_i)$ and $\mathcal{X}''(x_i^+) = \mathcal{X}'(x_i^+) = \mathcal{X}'(x_i) = \mathcal{X}(x_i) = v_i = \mathcal{X}_1(x_i)$.

Putting the above cases together, we have proved Proposition 4.

Finally, from Proposition 1 and 2, we can derive part 2 of Theorem 5.1 trivially, and we are done. □

Bibliography

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Tom M. Apostol. *Mathematical Analysis — A Modern Approach to Advanced Calculus*. Addison-Wesley, 1957.

[4] Gérard Berry. The constructive semantics of pure esteral (draft version 3). Draft Version 3, Ecole des Mines de Paris and INRIA, July 1999.

[5] Gérard Berry. *The Esterel v5 Language Primer Version 5.21 release 2.0*. Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, April 1999.

[6] 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.

[7] 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.

[8] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. *ACM SIGPLAN Notices*, 31(6):226–238, 1996.

[9] Richard Courant and Fritz John. *Introduction to Calculus and Analysis*, volume 1. Interscience Publishers, 1965.

[10] Antony Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of Symposium on Practical Aspects of Declarative Languages*. ACM, 2001.

[11] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.

[12] Anthony C. Daniels. *A Semantics for Functions and Behaviours*. PhD thesis, The University of Nottingham, December 1999.

[13] Abbas Edalat and Peter John Potts. A new representation for exact real numbers. *Electronical Notes in Theoretical Computer Science*, 6:14 pp., 1997. Mathematical foundations of programming semantics (Pittsburgh, PA, 1997).

[14] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, October 1997.

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

[16] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, Amsterdam, June 1997.

[17] 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`.

[18] 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.

[19] K. Hammond. Parallel functional programming: An introduction. In *International Symposium on Parallel Symbolic Computation*. World Scientific, 1994.

[20] 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.

[21] 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.

[22] Thomas A. Henzinger. The theory of hybrid automata. Technical report, University of California, Berkeley, 1996.

[23] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[24] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[25] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[26] Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[27] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[28] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 70–81, N.Y., September 27–29 1999. ACM Press.

[29] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Guy L. Steele Jr, editor, *Principles of Programming Languages*, volume 23, St Petersburg, Florida, 1996. ACM Press.

[30] Richard Kieburtz. Real-time reactive programming for embedded controllers. Available from author's home page, March 2001.

[31] Richard B. Kieburtz. Implementing closed domain-specific languages. In *[49]*, pages 1–2, 2000.

[32] O. (Oded) Maler, editor. *Hybrid and real-time systems: international workshop, HART '97, Grenoble, France, March 26–28, 1997: proceedings*, volume 1201 of *Lecture Notes in Computer Science*, New York, NY, USA, 1997. Springer-Verlag.

[33] Microchip Technology Inc. *PIC16C66 Datasheet*. Available on-line at `http://www.microchip.com/`.

[34] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[35] Edwin E. Moise. *Calculus.* Addison-Wesley, 1967.

[36] Andrzej S. Murawski and C.-H. L. Ong. Can safe recursion be interpreted in light logic? In *Second International Workshop on Implicit Computational Complexity*, Santa Barbara, June 200.

[37] Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48, 2000.

[38] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, continued. In *ACM SIGPLAN 2002 Haskell Workshop*, October 2002.

[39] Online documentation for the ICE programming language. `http://www.cognitoy.com/twiki/`.

[40] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[41] Izzet Pembeci, Henrik Nilsson, and Greogory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Principles and Practice of Declarative Programming (PPDP'02)*, October 2002.

[42] John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conference on Robotics and Automation*, 1999.

[43] Jonathan Rees and William Clinger (eds.). Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[44] 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.

[45] MindRover official web-site. `http://www.mindrover.com/`.

[46] RoboCup official web-site. `http://www.robocup.org/`.

[47] Christopher Strachey. The varieties of programming language. Technical Report Technical monograph PRG-10, Programming research group, Oxford University computing laboratory, 1973.

[48] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.

[49] Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.

[50] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, 1998.

[51] J. E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, 1990.

[52] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press U.K., London, 1985.

[53] 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.

[54] 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.

[55] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of Fourth International Symposium on Practical Aspects of Declarative Languages*. ACM, Jan 2002.