Sculthorpe, Neil (2011) Towards safe and efficient functional reactive programming. PhD thesis, University of Nottingham.

TOWARDS SAFE AND EFFICIENT FUNCTIONAL REACTIVE PROGRAMMING

NEIL SCULTHORPE, BSc.

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

July 2011

Abstract

Functional Reactive Programming (FRP) is an approach to reactive programming where systems are structured as networks of functions operating on time-varying values (signals). FRP is based on the synchronous data-flow paradigm and supports both continuous-time and discrete-time signals (hybrid systems). What sets FRP apart from most other reactive languages is its support for systems with highly dynamic structure (dynamism) and higher-order reactive constructs (higher-order data-flow). However, the price paid for these features has been the loss of the safety and performance guarantees provided by other, less expressive, reactive languages.

Statically guaranteeing safety properties of programs is an attractive proposition. This is true in particular for typical application domains for reactive programming such as embedded systems. To that end, many existing reactive languages have type systems or other static checks that guarantee domain-specific constraints, such as feedback being well-formed (causality analysis). However, compared with FRP, they are limited in their capacity to support dynamism and higher-order data-flow. On the other hand, as established static techniques do not suffice for highly structurally dynamic systems, FRP generally enforces few domain-specific constraints, leaving the FRP programmer to manually check that the constraints are respected. Thus, there is currently a trade-off between static guarantees and dynamism among reactive languages.

This thesis contributes towards advancing the safety and efficiency of FRP by studying highly structurally dynamic networks of functions operating on mixed (yet distinct) continuous-time and discrete-time signals. First, an ideal denotational semantics is defined for this kind of FRP, along with a type system that captures domain-specific constraints. The correctness and practicality of the language and type system are then demonstrated by proof-of-concept implementations in Agda and Haskell. Finally, temporal properties of signals and of functions on signals are expressed using techniques from temporal logic, as motivation and justification for a range of optimisations.

Acknowledgements

First and foremost I acknowledge my supervisor Henrik Nilsson. His guidance and assistance over the last five years have been immeasurable, and this thesis would not have been possible without him.

I also acknowledge the other members of the Functional Programming Laboratory, from whom I have learned much in my time at Nottingham. In particular: Graham Hutton who taught me functional programming, Thorsten Altenkirch who taught me logic, and Nils Anders Danielsson who taught me Agda. I also thank those who have provided me with feedback on this thesis or on my earlier papers, including Florent Balestrieri, Joey Capper, Laurence Day, and especially, George Giorgidze.

Finally, I thank my examiners Venanzio Capretta and Colin Runciman for their time and constructive comments.

Contents

List of Figures

Chapter 1

Introduction

For many application domains, systems are required to be *reactive* rather than *transformational* [50]. The input to such systems is not known in advance, but instead arrives continuously during execution. A reactive system is thus expected to interact with its environment, interleaving input and output in a timely manner [8, 9]. By *timely*, it is meant that a response is expected within an amount of time that is "reasonable" for the application domain at hand. Consequently, some reactive systems provide hard real-time guarantees (meeting deadlines is essential), while others, intended for less strict domains, only achieve soft real-time (meeting deadlines is desirable) [17, 128].

Functional Reactive Programming (FRP) grew out of Conal Elliott's and Paul Hudak's work on Functional Reactive Animation (Fran) [38]. The aim of FRP is to allow the full power of modern Functional Programming [11, 58, 122] to be used for implementing reactive systems. The basic idea is to model input and output as time-varying values called *signals*. Systems are then described by combining *signal functions* (functions mapping signals to signals) into *signal processing networks*. The nature of the signals depends on the application domain. Examples include: video streams in the context of animation [38], graphical user interfaces [25, 26], visual tracking [92, 103], and games [20, 27]; sensor input and control signals in robotics [57, 102] and animal monitoring [97] applications; and synthesised sound signals [43].

Compared to other reactive languages, FRP is characterised by being highly expressive, but lacking in safety guarantees and efficiency [105]. This thesis investigates ways of overcoming those two deficiencies without sacrificing the expressiveness of the FRP paradigm. Principally, this is achieved through a conceptual model designed to precisely characterise the abstractions of FRP, and a type system that captures FRP-specific constraints. Properties of this model, particularly those relating to notions of signal change with respect to time, are then studied as motivation and justification for a range of FRP-specific optimisations. The specific contributions of this thesis are given in Section 1.3. However, the wider context of this work first needs to be clarified.

1.1 Reactive Languages

A number of FRP variants exist. The basics of the early FRP approaches [38, 92, 127] are discussed in Chapter 3, and a number of others are overviewed in Chapter 10. However, the *synchronous data-flow principle* (modelling reactions as being instantaneous) [6, 48], and support for both *continuous* and *discrete* time (hybrid systems), are common to most FRP variants. There are thus close connections to synchronous data-flow languages such as Esterel [9, 10], Lustre [48], and Lucid Synchrone [104]; hybrid automata [52]; and languages for hybrid modelling and simulation, such as Simulink [2]. However, FRP goes beyond most of these approaches by supporting *highly dynamic system structure* and *higher-order reactive constructs*. A system is *structurally dynamic* if its structure can change over time. It is *highly* structurally dynamic if its structural configurations are not known in advance, but can be computed during execution. Throughout this thesis the term *dynamism* will be used to mean highly dynamic system structure. Likewise, the term *higher-order data-flow* will mean that some reactive constructs (signals or signal functions, depending on the FRP variant) are a first-class abstraction in the language.

Dynamism and higher-order data-flow are becoming ever more important aspects of reactive programming. They are essential for implementing reconfigurable systems, including systems that receive software updates whilst running, which are increasingly prevalent [22]. They also significantly extend the range of reactive systems that can be described naturally and easily. Examples include visual tracking [92], video games [20, 27], and virtual-reality applications [12].

However, the expressiveness granted by dynamism and higher-order data-flow comes at a cost: most FRP languages lack the space and time guarantees provided by most synchronous data-flow languages. The reason for this difference in the two paradigms stems mainly from their typical application domains. FRP has its origins in multimedia [38, 56], whereas the synchronous data-flow languages have had commercial success in safety-critical domains such as control systems for aeroplanes and nuclear power plants [7]. However, it is hard to draw a line between the two paradigms on this basis. There has been work both on extending synchronous data-flow languages with dynamism and higher-order features [17, 22, 120], as well as on restricted FRP languages that have guaranteed space and time bounds [128, 129].

The other main distinction between the two paradigms is their model of time. The synchronous data-flow languages are based on data-flow stream processing [126] (with time being represented by the ordering of values in the stream), and thus have an inherently discrete notion of time. By contrast, one of the original motivations of FRP was to model time continuously, leaving the implementation to automate discretisation [38]. However, this isn't a clear distinction either, as some FRP variants abandon the continuous model of time and adopt a stream-based approach directly [100, 128].

1.2 Embedded Domain-Specific Languages

FRP variants are usually implemented as *embedded domain-specific languages* (EDSLs) [55]. This involves defining the FRP language as a library within some general-purpose (and usually functional) host language. To date, the most common choice of host language for FRP has been *Haskell* [11, 122]. FRP variants embedded as Haskell libraries include: Fran [38], Yampa [90, 92], Reactive [37], Grapefruit [63] and Elerea [99, 100]. A notable exception is FrTime [23, 24], an FRP language embedded in the DrScheme environment [40].

There are many advantages to the embedded approach. Creating a programming language from scratch is a laborious task, whereas an EDSL can be implemented quickly and easily as only domain-specific aspects need to defined. General-purpose language features (e.g. numbers), as well as associated libraries and tools (e.g. debuggers), come for free. If several EDSLs are embedded in the same host language, then connecting them together is significantly simpler than connecting independent languages. Learning an embedded language is also much easier for an end-user who is already familiar with its host language. [55]

EDSLs also leverage the compilation and optimisation facilities of the host language. However, what does *not* come for free are domain-specific optimisations (or domain-specific error messages [51]), as a host-language compiler has no knowledge of the embedded domain. While it is possible to encode optimisations within an EDSL such that they will be applied at run-time [90], this is significantly less efficient than compile-time optimisation. This is a well known problem [28, 55], and many EDSLs come with either a pre-processor [77, 113] or domain-specific compiler [39, 72] to improve performance. One of the reasons that Haskell is considered a good host language is that the widely used Glasgow Haskell Compiler (GHC) [121] provides facilities for *compile-time meta-programming* in the form of *Template Haskell* [115] and *quasiquoting* [42, 80]. This is a convenient means of implementing such domain-specific optimisations [113], and also allows significant flexibility in terms of the domain-specific syntax that can be encoded. However, the highly dynamic nature of FRP means that some optimisation opportunities are not known statically, and only arise at run-time. Chapter 8 of this thesis is concerned with identifying domain-specific properties of FRP that could be exploited for domain-specific optimisation.

A drawback of EDSLs is that their type systems are restricted by those of their hosts. By comparison, a stand-alone language can have a type system specialised to its application domain. Furthermore, a domain-specific compiler can check for additional domain-specific constraints beyond those expressed by the types [21, 30]. Such static checks are important, as the increasing complexity of reactive systems makes it correspondingly harder to test them sufficiently thoroughly. Moreover, in many typical reactive applications, such as embedded systems [69], the cost of failure is very high, thereby making it imperative to statically guarantee that the system will not fail [7]. Fortunately for the embedded approach, the type systems of general-purpose functional languages are becoming ever-more expressive [125, 130], making it increasingly feasible to express domain-specific constraints within the type system of the host language. Chapters 7 and 9 of this thesis are concerned with this style of capturing FRP-specific constraints.

Two host languages are used in this thesis: Haskell, and the dependently typed language *Agda* [95]. Haskell is chosen because it has repeatedly proved itself a practical and effective host language for FRP. The additional choice of Agda as a host language serves two purposes. First, because it has a more powerful type system, it is easier to encode domain-specific constraints in Agda than in Haskell. This makes Agda more suitable for prototyping FRP type systems. Second, Agda provides totality and termination checks. An embedding of FRP in

Agda is therefore guaranteed to be total and terminating, and thus the embedding constitutes a machine-checked proof of the safety of the FRP implementation. Finally, note that Haskell and Agda are both purely functional languages, and syntactically very similar. This maintains a correspondence between the two embeddings and makes it fairly easy (in this instance) to translate code between the two.

1.3 Contributions and Thesis Structure

The work presented in this thesis is mostly drawn from three previous papers [110, 111, 112] co-authored by myself and Henrik Nilsson (my supervisor). This thesis supersedes all three papers, refining and improving on the work therein, as well as adding some additional material.

The motivation behind this work was that implementing FRP (in its full expressiveness) in a way that scales efficiently has proved challenging [36, 76], and remains an active research area [37, 63, 75, 100]. Additionally, dynamism and the embedded approach to implementation have obstructed many of the static checks present in synchronous data-flow languages [128]. As a step towards overcoming these issues, we defined a new conceptual FRP model that respects the abstractions of the FRP domain, while being convenient to implement, optimise, program with, and reason about. This model, which we call N-ary FRP, is based around *signal functions* as the central reactive abstraction, while supporting *multi-kinded* signals (maintaining a conceptual distinction between continuous-time and discrete-time signals). While neither signal-function–based FRP nor multi-kinded signals are new ideas, to our knowledge the combination of the two has not been studied before (other signal-function–based models typically embed discrete signals within continuous signals). I argue that such a model has many advantages including conceptual precision, ease of implementation, language safety, and optimisation opportunities.

In brief, the contributions of this thesis are:

- An overview and comparison of the signal based and signal-function–based FRP models.
- A new FRP language based on signal functions and multi-kinded signals (N-ary FRP).
- An idealised denotational semantics for N-ary FRP, formally encoded in Agda.
- Type-system refinements for N-ary FRP that guarantee totality without prohibiting *feedback* and *uninitialised signals*.
- Proof-of-concept embeddings of N-ary FRP in Haskell and Agda, including the type-system refinements.
- Identifying and formalising a number of useful domain-specific properties of the N-ary FRP model, in particular those pertaining to notions of signal change and change-based optimisation.

The structure of the thesis, along with the specific contributions of this author, is as follows:

- Chapter 2 contains a brief introduction to the Agda language and to the notational conventions used in this thesis. There is no technical contribution in this chapter.

- Chapter 3 introduces the fundamental concepts of FRP, gives some examples of FRP programming, and compares the signal-based and signal-function–based models. The comparison is based on an earlier version written jointly with Henrik Nilsson [110]. The concepts in this chapter are not new, and a similar comparison discussing some (though not all) of the same issues can be found in Courtney [25].

- Chapter 4 defines the N-ary FRP conceptual model and language, and gives some examples of N-ary FRP programming. The technical content of this chapter is based on joint work with Henrik Nilsson, and has appeared previously in [110].

- Chapter 5 describes an embedding of the N-ary FRP language in both Agda and Haskell. This is individual work of this author, and is based on an earlier version that accompanied [112].

- Chapter 6 contains an introduction to temporal logic, an encoding of temporal logic in Agda, and the formulation of some N-ary FRP properties using this encoding. This is individual work of this author, and is a significant revision of an earlier version that appeared in [110].

- Chapter 7 extends the N-ary FRP language with a feedback combinator, and refines the N-ary FRP type system to enforce the domain-specific constraint that all reactive feedback must be well-defined. This chapter also contains corresponding extensions of the Agda and Haskell embeddings from Chapter 5. This is individual work of this author, a preliminary version of which appeared in [112].

- Chapter 8 considers optimisation opportunities for FRP, and formalises and proves several domain-specific properties of N-ary FRP (using the temporal logic from Chapter 6) that could be exploited for domain-specific optimisation. This is individual work of this author, and is a significant revision of earlier versions that appeared in [110, 111].

- Chapter 9 discusses experimental extensions to the N-ary FRP model. The N-ary FRP language is extended with uninitialised signals, and the type system is correspondingly refined to ensure that this is safe. An extension of the N-ary FRP type system that allows a more precise causality analysis is also considered. This is individual work of this author. Most of this chapter is previously unpublished, though an earlier version of the uninitialised-signals extension appeared in [112].

- Chapter 10 overviews related work. This is partially based on a previous review of recent FRP developments written jointly with Henrik Nilsson [110].

- Chapter 11 discusses avenues for future work.

Finally, note that all definitions and program code in this thesis have been formalised in Agda by this author. All such Agda code, along with some supplementary proof scripts and the Agda and Haskell implementations of N-ary FRP, are available in an online archive [1].

Chapter 2

Agda and Notation

As discussed in Section 1.2, this thesis contains embeddings of FRP in two languages: Haskell and Agda. However, a meta-language for expressing the *semantics* of FRP is also required. The meta-language used by this thesis is Agda augmented with some additional syntax. This language will be referred to as *AgdaFRP*. However, the contents of this thesis have also been encoded in "genuine" Agda (the code is available in the online archive [1]), and translation between the two is mostly straightforward. AgdaFRP is also the language used to express example FRP code throughout this thesis, except when working within a specific host language.

This chapter gives a basic introduction to Agda, and then describes how AgdaFRP differs. A reader acquainted with Agda can skip to Section 2.2.

2.1 Introduction to Agda

This section introduces the features of Agda that are pertinent to this thesis. The reader is assumed to have a basic familiarity with Haskell; an unfamiliar reader should consult an introductory textbook such as Bird [11], Hudak [56] or Hutton [60]. For a more comprehensive introduction to Agda, consult Norell [96].

2.1.1 Overview

Agda is a *dependently typed* [94] language. The essence of dependent types is that the *type* of the result of a function may depend on the *value* of its argument. In Agda there is little distinction between data and types, with both appearing in type signatures and in program code. To ensure that type checking is decidable in such a setting, Agda requires all functions to be *total* and *terminating*. This also guarantees that Agda programs are free of run-time errors[1].

Together, these features mean that Agda can exploit the Curry-Howard Correspondence [31, 54] to encode propositions as types. The basic idea is that a type can represent a proposition, with the elements of that type being proofs of the proposition. Thus any inhabited type is a true proposition, and any uninhabited type is a false proposition. Properties about Agda programs

[1]Note however that Agda itself has not been formally verified, so all claims of guarantees in this thesis depend on the assumption that the Agda language and type-checker are error-free.

can thus be proved within the Agda language, and furthermore these proofs can be used in Agda programs to ensure totality and termination.

2.1.2 Data Types

Agda data types are defined in a similar manner to GADTs [64] in Haskell. For example:

```
data Unit : Set where
  unit : Unit
data Bool : Set where
  false : Bool
  true  : Bool
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

Type signatures are given with a single colon. *Set* is the type of types, analogous to kind $*$ in Haskell. Note that throughout this thesis, data constructors will be type-set in a sans-serif roman font, and keywords in **boldface roman** font.

Parametrised data types are also possible. For example, option types, lists, product types and sum types are defined as follows:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A
data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
data _⊎_ (A B : Set) : Set where
  inl : A → A ⊎ B
  inr : B → A ⊎ B
```

The underscores are used to define infix (and more generally, mixfix) operators, with the position of the underscores denoting the position of the arguments. The types of the parameters to these data types are stated explicitly as parameters may have types other than *Set* (though in these particular cases they could be inferred automatically).

Data types may also have *indices* in addition to parameters. The distinction is that a parameter is fixed over all constructors, whereas an index may depend on the constructor. Syntactically, a parameter appears before the colon in the type signature, and an index appears after the colon. For example, the type of vectors (lists indexed by their length) is as follows:

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A zero
  _::_  : {n : ℕ} → A → Vec A n → Vec A (succ n)
```

A is a *parameter* and thus is the same for all constructors, whereas the natural number denoting the length of the vector is an *index* and varies between constructors. Note that Agda allows constructor names to be overloaded, and thus the same constructors can be used as for *List*. The $\{n : \mathbb{N}\}$ will be explained in the next section.

Finally, a dependent product type (Σ-type) is defined as follows:

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (a : A) → B a → Σ A B
```

This has a *dependent* type; specifically the *type* of the second element of the product depends on the *value* of the first element. The notation $(a\ :\ A)$ means that the value of type A is bound to identifier a for the remainder (to the right) of the type signature. $B\ a$ is a *type* that is computed by applying the type constructor B to the value a. There will be an example of using Σ-types in the next section.

2.1.3 Functions

Functions are defined in a similar manner to Haskell; for example:

```
isZero : ℕ → Bool
isZero zero     = true
isZero (succ _) = false
```

However, when defining polymorphic functions all type arguments must be explicitly quantified. For example, a polymorphic identity function could be defined as follows:

```
id : (A : Set) → A → A
id A a = a
```

In many cases, some arguments can be automatically inferred at the application site. To exploit this, arguments can be denoted as *implicit arguments* by enclosing them in curly braces. For example, the *Set* argument of the *id* function could be made implicit as follows:

```
id : {A : Set} → A → A
id a = a
```

Whenever *id* is used, providing the *Set* argument is optional if it can be inferred from the context. Thus, for example, the following two definitions are equivalent:

```
idBool : Bool → Bool
idBool = id

idBool : Bool → Bool
idBool = id {Bool}
```

If implicit arguments are needed in the function definition they can be brought into scope explicitly like so:

```
id : {A : Set} → A → A
id {A} a = a
```

When the result type of a function is not fully known, Σ-types can be used to encode existential quantification. For example, a function that maps a list to a vector can be defined as follows:

```
listToVec : {A : Set} → List A → Σ ℕ (Vec A)
listToVec [] = (zero, [])
listToVec (a :: as) with listToVec as
... | (n, v) = (succ n, (a :: v))
```

Note that **with** is analogous to Haskell's **case**: in this instance the result of *listToVec as* is bound to the pattern (n, v).

2.1.4 Propositions and Proofs

As previously mentioned, propositions can be represented as types and proofs can be represented as elements of those types. Falsehood is represented by an empty type (it is uninhabited and is therefore unprovable), and truth by the unit type (it is inhabited by unit and is therefore always true):

```
True : Set
True = Unit

data False : Set where
```

Implication corresponds to a function. Thus, for example, *Not A* can be defined as a function mapping *A* to *False*:

```
Not : Set → Set
Not A = A → False
```

Note that the expression *A → False* is a *type*: the arrow is the same function arrow that appears in type signatures. Somewhat unfortunately, the same arrow symbol is used for lambda abstractions (in the same way as Haskell). Thus, in an equivalent definition of *Not* that uses a lambda, the same symbol has two different meaning in the same expression:

```
Not : Set → Set
Not = λ A → (A → False)
```

True, *False* and *Not* are useful for defining predicates over data types, for example:

```
IsNothing : {A : Set} → Maybe A → Set
IsNothing nothing = True
IsNothing (just _) = False

IsJust : {A : Set} → Maybe A → Set
IsJust ma = Not (IsNothing ma)
```

However, many propositions require their own specialised data type. For example, propositional equality can be defined as follows:

```
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a
```

Thus refl (reflexivity) is the sole proof of propositional equality.

A more complicated proof data type is that of the less-than relation on natural numbers:

```
data _<_ : ℕ → ℕ → Set where
  zlt : ∀ {n}             → zero   < succ n
  slt : ∀ {m n} → m < n → succ m < succ n
```

That is, zero is less than the successor of any natural number, and succ is monotonic with respect to $<$. The use of ∀ is Agda syntactic sugar that allows the types of identifiers to be omitted when they can be inferred. For example, $\forall\ \{m\ n\}$ is sugar for $\{m\ n\ :\ \mathbb{N}\}$. This notation is also valid for explicit arguments, in which case the curly braces are omitted.

New properties/types can be defined in the same way as functions. For example, the less-than-or-equal relation can be defined as follows:

```
_⩽_ : ℕ → ℕ → Set
m ⩽ n = (m < n) ⊎ (m ≡ n)
```

Properties can be used in function definitions to ensure that functions are *total*. For example, consider the following subtraction function:

```
sub : (m n : ℕ) → (n ⩽ m) → ℕ
sub m        zero     p            = m
sub zero     (succ n) (inl ())
sub zero     (succ n) (inr ())
sub (succ m) (succ n) (inl (slt p)) = sub m n (inl p)
sub (succ m) (succ .m) (inr refl)   = zero
```

This function takes a proof that $(n \leqslant m)$ as an additional argument, thereby constraining the natural numbers that *sub* can be applied to. Note that this is a dependent function: the type of the proof depends on the values of the first two arguments. Two as-yet-unmentioned features of Agda are used in this definition. First, the use of () is an *absurd pattern*. This is special syntax used to denote that the type is empty, and thus that there is no possible match for this pattern. In this particular case, there are no constructors of type (succ $n \equiv$ zero) or of type (succ $n <$ zero). When an absurd pattern is used, the right-hand side of the equation is omitted (as it cannot be reached). Using absurd patterns is necessary to convince Agda that functions are total; these cases cannot just be omitted as would be done in Haskell. The second feature is the *dot-pattern* in the final case. When the refl constructor is pattern matched on, its type unifies the two identifiers m and n. This is denoted by replacing n with $.m$ (or m with $.n$), which means that n has been constrained to be equal to m.

2.2 AgdaFRP

The meta-language of this thesis is an Agda variant that will be referred to as AgdaFRP. This is not a language with any formal basis: it is merely a convenient notation for expressing the ideas in this thesis without straying too far from the accompanying Agda encoding. There are three main reasons why Agda is not used directly:

- Some conceptual definitions (particularly those pertaining to real numbers) are not computable, and thus cannot be defined in Agda.
- Agda requires all functions to be total and terminating: while these are desirable features of an FRP language, it is useful to state definitions where this is not the case (particularly when partiality or termination is the subject of discussion).
- Augmenting Agda with additional syntactic sugar allows the presentation to be clarified, as some of the Agda definitions are rather verbose.

In the first case, AgdaFRP makes use of more general mathematical notation to express non-computable conceptual definitions. In the accompanying Agda code, these definitions are postulated as axioms.

In the second case, the definitions have still been encoded in Agda: the termination checker is simply switched off for those definitions (using the *--no-termination-check* option). Note that some defined functions are terminating, but are not recognised as such by the Agda termination checker. These functions could be restructured in such a way that they pass the termination checker, but that would make their presentation less clear. Also, AgdaFRP permits infinite terms (similarly to lazy languages such as Haskell). This is necessary for some of the example code from previous FRP variants, and is also used once when defining N-ary FRP code (discussed in Section 4.4).

In the third case, the additional syntactic sugar is mostly taken from Haskell. Specifically, the following conveniences are permitted:

- Operator sections in the style of Haskell.
- Pattern guards, **case** expressions, and pattern matching under lambdas and in **let** expressions. In the accompanying Agda code these are all replaced by **with** expressions.
- Implicit arguments may be omitted from type signatures. Any free identifiers present in type signatures should be assumed to be universally quantified at the top level (as is the case in Haskell).
- Implicit arguments that can be inferred by the reader are omitted even if Agda cannot infer them.
- Some function names are overloaded when it is always clear from the context which one is meant. For example, $\leqslant$ is used both as a type constructor (as in Section 2.1.4) and as a binary operator returning a Boolean. Furthermore, it is overloaded onto several numeric types.

Agda also has a *universe hierarchy*: the type of *Set* is Set_1, the type of Set_1 is Set_2, and so forth. However, for simplicity, AgdaFRP takes the type of *Set* to be *Set*. The same approach is taken in the accompanying Agda code by using the *--type-in-type* option. This creates an inconsistency in the logic, but this inconsistency is not exploited.

Finally, note that AgdaFRP is used for expressing both FRP semantics and example FRP code. To limit confusion between semantics, example FRP code, and embedded Agda implementations, this thesis uses the $\approx$ symbol instead of $=$ when defining the *semantics* of an entity that is first-class in the FRP language.

Chapter 3

Functional Reactive Programming

This chapter introduces the fundamental concepts of Functional Reactive Programming (FRP). Two distinct models of FRP are then described: Classic FRP (CFRP) and Unary FRP (UFRP). This should give background on FRP, and also provide some examples of FRP programming. However, as these particular FRP models are not the principal topic of this thesis, only selected parts of each variant are discussed. The chapter concludes with a comparison of these two branches of FRP, as motivation for the N-ary FRP model that will follow in Chapter 4.

3.1 Why *Functional* Reactive Programming?

Taking a purely functional approach to reactive programming brings with it all of the usual advantages of functional programming, such as powerful facilities for modularity, abstraction and ease of reasoning [58]. However, there are some aspects of reactive programming for which a functional approach is particularly well-suited.

Some common applications of FRP include modelling physical systems [89], or involve simulating physical laws as part of a larger system (such as governing the movement of entities in a video game [27]). Such physical laws are often defined by differential equations, and it is much easier for an end-user to translate such equations into declarative code than into imperative code [89, 127].

One advantage of a *purely* functional approach is that it makes the system easier to parallelise and distribute. Consider that FRP programs define synchronous data-flow networks that execute over a time period. Each network node encapsulates a local state that allows it to remember the past, but each such state is inaccessible from outside the node. Purely functional data types are exactly what is needed to represent such nodes, as, being immutable, they cannot be modified from elsewhere in the program. Thus if two nodes (or sub-networks) are in parallel in the network structure, then it is safe to execute them concurrently, as they cannot interfere with each other. As demonstrated by Google's MapReduce Framework [34], once such non-interference is established it is much easier to efficiently execute extremely large programs (whether on a multi-processor machine or in a distributed setting).

If a reactive language is implemented as a domain-specific embedding (as discussed in Section 1.2), then it is highly desirable that the host language provides sufficient abstraction facilities to express all the primitives of the reactive language [55]. Not only does this make implementing an embedding much simpler, it also allows the end user to be presented with an uncluttered interface. Powerful abstraction capabilities may not be unique to functional programming, but functional languages typically have more powerful abstraction capabilities (such as first-class functions) than most imperative languages. In particular, it is much easier to provide abstractions if it is possible to safely name any expression in the host language (as is the case in a pure lazy language or a pure total language).

None of the FRP variants discussed in this thesis are dependently typed languages. Yet the two host languages used in this thesis are Agda, which is dependently typed, and Haskell, which provides some dependently typed features via language extensions (discussed in Section 5.3.1). This is deliberate: the N-ary FRP type system could not be directly embedded in a simply typed host language. Note however that this requirement is specific to N-ary FRP; most FRP variants have been successfully embedded in simply typed languages. Of course, if an FRP language *is* embedded in a dependently typed language, then the FRP programmer can incorporate dependent types into her FRP programs. In this author's opinion, dependent types would be as useful for capturing invariants and providing greater type precision in FRP as they are in general, but being in a reactive setting does not make them *more* useful than usual.

3.2 FRP Fundamentals

FRP languages can be considered to have two levels to them: a *functional* level and a *reactive* level [128]. The functional level is a pure functional language. FRP implementations are usually embedded in a host language, and in these cases the functional level is provided entirely by the host. The reactive level is concerned with time-varying values called *signals*. At this level, functions operating on signals are used to construct synchronous data-flow networks. There are thus two distinct function spaces, which allows for level-specific operations. However, the levels are interdependent. The reactive level relies on the functional level for carrying out arbitrary pointwise computations on signals, while some reactive constructs are first-class entities at the functional level.

An FRP language consists of a set of primitive first-class reactive constructs, and a set of primitive combinators that combine reactive constructs into signal processing networks. The key point about such primitives is that they only allow the construction of networks that respect the conceptual model.

3.2.1 Continuous-Time Signals

Time is considered to be continuous in FRP. Signals are thus modelled as functions from continuous-time to value, where time is taken to be the set of non-negative real numbers:

$$\mathit{Time} \approx \{t \in \mathbb{R} \mid t \geqslant 0\}$$
$$\mathit{Signal}\ A \approx \mathit{Time} \rightarrow A$$

This conceptual model provides the foundation for an ideal denotational semantics. Of course, in order to be reactive, any digital implementation of a continuous-time signal will have to sample the signal over a discrete sequence of time steps, and will consequently only approximate the ideal semantics. The advantage of the conceptual model is that it abstracts away from such implementation details. It makes no assumptions as to the rate of sampling, whether the sampling rate is fixed, or how sampling is performed [38]. It also avoids many of the problems of composing subsystems that have different sampling rates. The ideal semantics is thus helpful for understanding FRP programs, at least to a first approximation. It is also abstract enough to leave FRP implementers considerable freedom [36, 37].

That said, implementing FRP completely faithfully to the ideal semantics is challenging. At the very least, a faithful implementation should, for "reasonable programs", converge to the ideal semantics in the limit as the sampling interval tends to zero [127]. But even then it is hard to know how densely one needs to sample before an answer is acceptably close to the ideal.

3.2.2 Signal Functions

Signal functions are conceptually functions on signals:

$$SF\ A\ B \approx Signal\ A \rightarrow Signal\ B$$

In some FRP languages (such as Yampa [92]), signal functions, rather than signals, are the primary reactive abstraction. Signal functions are first-class entities in such languages, while signals have no independent existence of their own. This is the approach taken by the N-ary FRP language defined in this thesis (Chapter 4).

What if plain signals are needed; that is, a time-varying value that depends on no input? Well, a signal function that takes a unit signal as input (or is entirely polymorphic in its input) essentially serves the same purpose. (However, see the discussion in Section 3.2.5: these are really *signal generators*.)

To make signal functions suitable for implementing reactive systems, signal functions are constrained to be *temporally causal*. Temporal causality means that *effects* must not precede *causes* with respect to time (the present can depend on the past but not the future). Thus, a temporally causal signal function is one such that its output at time t is uniquely determined by its input over the interval $[0, t]$. This is formalised in Section 6.5.2. In all FRP variants that are considered in this thesis, temporal causality is enforced by only providing primitive signal functions that are temporally causal and primitive combinators that preserve temporal causality.

Aside: Notions of Causality

Temporal causality is specified because there are other notions for which the term causality is used. *Computational causality* refers to cause and effect relationships where one thing is computed from another, but without reference to any notion of time. In the field of modelling languages the term *causality* is used in this computational sense. Thus, in such languages, a *causal* model is one defined by directed equations (what is computed from what is explicit), whereas a *non-causal* (or *acausal*) model is one defined by undirected equations. In the latter case, the equations express a relation between signals, but do not divide them into inputs and outputs. *Causalisation* of such models means re-writing the equations in a computationally causal form, which is required to make them suitable for simulation. If there is no dependency whatsoever between two signals, then they are said to be *causally unrelated*. [19]

FRP programs are computationally causal specifications of temporally causal systems. The generalisation of FRP to computationally acausal specifications is called Functional Hybrid Modelling [93], but that is beyond the scope of this thesis. Finally, note that the term *temporally acausal* refers to situations where the present can depend on the future as well as the past, and that the term *temporally anticausal* refers to situations where the present can depend on the future but not the past. For the remainder of this thesis, whenever the terms *causal* or *acausal* are used, it will be in the temporal sense.

3.2.3 Discrete-Time Signals

Conceptually, *discrete-time signals* (often called *event signals*) are signals whose domain of definition is an at-most-countable set of points in time. Each point of definition signifies some event that is without any extent in time. Inclusion of discrete-time signals, along with operations on them and operations for mediating between continuous-time and discrete-time signals, is what makes most FRP variants capable of handling hybrid systems [38, 92].

However, different FRP variants have taken different approaches to the nature of discrete-time signals. One possibility is to make a fundamental distinction between continuous-time and discrete-time signals on the grounds that they enjoy distinct properties [24, 37, 38]. Separating them facilitates taking advantage of these differences for being more precise about applicable operations or for optimisation purposes [37, 63]. This approach will be referred to as *multi-kinded* FRP as there is more than one kind of signal. For example, CFRP is multi-kinded (see Section 3.3.1).

Another possibility is to define discrete-time signals as a subtype of continuous-time signals by lifting the range of signals, such as by using an option type [92, 99, 128]. This approach will be referred to as *single-kinded* FRP as there fundamentally is only one kind of signal. For example, UFRP is single-kinded (see Section 3.4.1).

3.2.4 Structural Dynamism

As discussed in Section 1.1, one of the main things that sets FRP apart from the synchronous data-flow languages is its support for dynamism. Dynamic reconfigurations of the network structure are referred to as *structural switches*, and the points in time at which reconfiguration takes place are called *moments of switching*.

A common way that FRP languages allow dynamism to be expressed is by providing one or more *switching combinators* as language primitives. As structural switches are discrete instantaneous occurrences, event signals are used to control when they occur. Typically, a switching combinator will be controlled by a specific event signal. Most combinators are such that they apply a structural switch either at the time of the *first* event occurrence within the event signal, or *whenever* there is an event occurrence within the event signal. The details of switching varies between FRP systems, but the essential idea is that, at the moment of switching, one signal, called the *subordinate* signal, is removed from the network, and a new signal, called the *residual* signal, is inserted in its place. This is called *switching-out* the subordinate signal and *switching-in* the residual signal.

Switching combinators often allow the residual signal to depend on the event that triggered the switch. This is usually expressed by a *switching function*: a function mapping the event value to a signal. This means that, in general, the residual signal cannot be computed until the moment of switching. This has important consequences. First, it cannot be assumed that switching only happens within a predetermined finite set of system configurations. Second, it raises the question as to over what range of time the residual signal is defined: from the system start time or from the time it was switched-in? This is discussed in the next section.

For a concrete example of a switching combinator, see Section 3.3.3 that provides a formal definition of such a combinator in the setting of CFRP.

What about a setting where signal functions, not signals, is the primary reactive abstraction? In that case, switching takes place between signal functions, not signals. Other than that, the ideas are very similar. See Section 3.4.4 for a definition of that style of switching combinator.

3.2.5 Signal Generators

Switching combinators defined on signals can either "start" the residual signal at the same time as the subordinate signal, or when it is switched-in. If *all* switching combinators adhere to the former, then the start time of *all* signals in the entire system will be the system start time. This is the approach taken by Fran [38] and Reactive [37].

The first choice is problematic if the residual signal depends on the value of the triggering event, as this is not known until the moment of switching. Consequently, at the moment of switching, the residual signal has to be retroactively computed up to that moment. In an implementation, this requires all past input to be remembered, a so-called *space leak*, and a catch-up computation to be performed, a so-called *time leak* [25, 36, 75, 76]. The longer the up-time of the system, the more cumbersome this becomes. Consequently, many FRP variants with first-class signals choose the second option: to start the residual signal at the moment of switching [99, 100, 127].

However, once there are signals that can start at different times, the conceptual model of signals as functions from time to value is insufficient. The value of a signal no longer just depends on the time at which it is sampled, but also the time at which it starts. To express this, the concept of a *signal generator* [63, 100, 127] is needed:

$$\begin{aligned} \mathit{StartTime} &= \mathit{Time} \\ \mathit{SampleTime} &= \mathit{Time} \end{aligned}$$

$$\mathit{SignalGenerator}\ A \approx \mathit{StartTime} \rightarrow \mathit{SampleTime} \rightarrow A$$

Or, equivalently, a signal generator is a function that, given a start time as an argument, produces a signal as the result:

$$\mathit{SignalGenerator}\ A \approx \mathit{StartTime} \rightarrow \mathit{Signal}\ A$$

The key point is that two signals created from the same signal generator can be (and often are) different if started at different times.

3.3 Classic FRP

There are several variants of CFRP, but they are all based around multi-kinded first-class signals: *Behaviours* (continuous-time signals) and *Events* (discrete-time signals). This section introduces a basic CFRP language, and then gives some examples of CFRP programming.

3.3.1 Behaviours and Events

In many CFRP variants, *Behaviours* and *Events* are actually *signal generators*, not signals [127]. In these cases, a *Behaviour* is thus (conceptually) a function that maps a start time and a sample time to a value:

$Behaviour\ A \approx StartTime \rightarrow SampleTime \rightarrow A$

An *Event* is similar, except that it produces a (time-ordered and finite) list of event occurrences *up to* the sample time[1]:

$Event\ A \approx StartTime \rightarrow SampleTime \rightarrow List\ (Time \times A)$

When defining similar functions over *Behaviours* and *Events*, this thesis adopts the naming convention of adding a 'B' or 'E' suffix, respectively. In most implementations some form of overloading is employed.

3.3.2 CFRP Primitives

This section introduces some CFRP primitives, along with their conceptual definitions. The utility functions used in these definitions can be found in Appendix A.

First, some *lifting functions* that lift values and functions from the functional level to operate over *Behaviours* are defined in the following pointwise fashion:

$constant : A \rightarrow Behaviour\ A$
$constant\ a \approx \lambda\ t_0\ t_1 \rightarrow a$

$liftB : (A \rightarrow B) \rightarrow Behaviour\ A \rightarrow Behaviour\ B$
$liftB\ f\ beh \approx \lambda\ t_0\ t_1 \rightarrow f\ (beh\ t_0\ t_1)$

$liftB2 : (A \rightarrow B \rightarrow C) \rightarrow Behaviour\ A \rightarrow Behaviour\ B \rightarrow Behaviour\ C$
$liftB2\ f\ beh_1\ beh_2 \approx \lambda\ t_0\ t_1 \rightarrow f\ (beh_1\ t_0\ t_1)\ (beh_2\ t_0\ t_1)$

It can be useful to ignore event occurrences at the start time of an *Event*; a primitive combinator that does this is defined as follows:

$notYet : Event\ A \rightarrow Event\ A$
$notYet\ ev \approx \lambda\ t_0\ t_1 \rightarrow dropWhile\ ((\leqslant\ t_0) \circ fst)\ (ev\ t_0\ t_1)$

A real-valued *Behaviour* can be integrated with respect to time. Note that the value of the resultant *Behaviour* at any given time depends upon the past values of the argument *Behaviour*:

$integral : Behaviour\ \mathbb{R} \rightarrow Behaviour\ \mathbb{R}$
$integral\ beh \approx \lambda\ t_0\ t_1 \rightarrow \int_{t_0}^{t_1} (beh\ t_0\ t)\ dt$

It is also useful to have an integration function that has an initial value other than zero. Such an *initialised integral* can be defined within the CFRP language (rather than as a primitive):

[1]There are also some side conditions on the definition of *Event*, but discussion of these is postponed until Section 4.1.5 to avoid obfuscating the key concepts in the present section.

$$\begin{aligned} &\mathit{iIntegral} : \mathbb{R} \rightarrow \mathit{Behaviour}\ \mathbb{R} \rightarrow \mathit{Behaviour}\ \mathbb{R} \\ &\mathit{iIntegral}\ x = \mathit{liftB}\ (+\ x) \circ \mathit{integral} \end{aligned}$$

This thesis adopts the naming convention of adding an 'i' prefix to functions that take an initial value as an argument.

Finally, *when* is a primitive function that mediates between *Behaviours* and *Events*:

$$\mathit{when} : (A \rightarrow \mathit{Bool}) \rightarrow \mathit{Behaviour}\ A \rightarrow \mathit{Event}\ A$$

The conceptual definition of *when* is omitted as it is quite involved. It can be found in Wan and Hudak [127]. Informally, the resultant *Event* contains an occurrence at each time point that the predicate (the first argument) applied to the value of the *Behaviour* (the second argument) changes from false to true. The value of the occurrence is the value of the *Behaviour* at that time point. There are two crucial points here: events only occur when the result of the predicate *changes* (not whenever it holds), and, consequently, there is never an event occurrence at the start time of *when*.

3.3.3 Switching between Behaviours

As discussed in Section 3.2.4, switching combinators are a means of expressing dynamism. The following is a typical CFRP switching combinator:

$$\begin{aligned} &\mathit{untilB} : \mathit{Behaviour}\ A \rightarrow \mathit{Event}\ E \rightarrow (E \rightarrow \mathit{Behaviour}\ A) \rightarrow \mathit{Behaviour}\ A \\ &\mathit{untilB}\ \mathit{beh}\ \mathit{ev}\ f \approx \lambda\ t_0\ t_1 \rightarrow \mathbf{case}\ \mathit{ev}\ t_0\ t_1\ \mathbf{of} \\ &\qquad [\,] \rightarrow \mathit{beh}\ t_0\ t_1 \\ &\qquad (t_e, e) :: \_ \rightarrow (f\ e)\ t_e\ t_1 \end{aligned}$$

The first argument is the subordinate *Behaviour*; the second argument is the *Event* that controls when the structural switch occurs; and the third argument is the switching function. The resultant *Behaviour* is that of the subordinate *Behaviour* until the first occurrence in the event signal. At which point, the switching function is applied to the event value to generate a residual *Behaviour*. Henceforth (including the moment of switching), the resultant *Behaviour* is that of the residual *Behaviour*, which only starts at the moment of switching.

Recall that an alternative design choice would be to have the residual *Behaviour* start at the same time as the subordinate *Behaviour*. The semantics of such a switching combinator would be:

$$\begin{aligned} &\mathit{untilB}' : \mathit{Behaviour}\ A \rightarrow \mathit{Event}\ E \rightarrow (E \rightarrow \mathit{Behaviour}\ A) \rightarrow \mathit{Behaviour}\ A \\ &\mathit{untilB}'\ \mathit{beh}\ \mathit{ev}\ f \approx \lambda\ t_0\ t_1 \rightarrow \mathbf{case}\ \mathit{ev}\ t_0\ t_1\ \mathbf{of} \\ &\qquad [\,] \rightarrow \mathit{beh}\ t_0\ t_1 \\ &\qquad (t_e, e) :: \_ \rightarrow (f\ e)\ t_0\ t_1 \end{aligned}$$

If *all* switches were of the *untilB′* type, then t_0 would always be 0, the global system start time. This means that the start time parameter becomes redundant, and *Behaviour* and *Event* become signals as opposed to signal generators. But, as has been discussed, this can lead to performance problems.

3.3.4 Example: Modelling Bouncing Balls

To demonstrate CFRP programming, the classic Bouncing-Ball example (also found in Nilsson [89], Courtney [25] and Liu [75]) is considered. Bouncing balls require hybrid modelling because the continuous motion of the ball is broken by discrete events (when the ball hits the ground). For simplicity: balls are modelled as point masses, an absence of air resistance is assumed, and only one dimension is considered (the height of the ball above the ground). However, to demonstrate the modularity and higher-order benefits of FRP, distinct balls that behave differently when they impact the ground will be considered.

Falling Balls

The configuration of a ball can be represented by a pair of its height and velocity:

```
Acceleration = ℝ
Velocity = ℝ
Height = ℝ
Ball = Height × Velocity
```

For the purposes of this example, assume the units are metres and seconds. The gravitational constant can thus be set:

```
g : Acceleration
g = 9.81
```

A *Behaviour* that models a freely falling ball can now be constructed. This is achieved by integrating the acceleration (in this case gravity) to compute the velocity, and integrating the velocity to compute the height. The *Behaviour* is parameterised on an initial ball configuration:

```
fallingBall : Ball → Behaviour Ball
fallingBall (h₀, v₀) = let a = constant (−g)
                           v = iIntegral v₀ a
                           h = iIntegral h₀ v
                       in liftB2 (,) h v
```

Bouncing Balls

The next step is to model interaction with the ground. First, a predicate to detect when a ball impacts the ground is required, as is a function to negate a ball's velocity:

```
detectImpact : Ball → Bool
detectImpact (h, _) = h ⩽ 0

negateVel : Ball → Ball
negateVel (h, v) = (h, −v)
```

Next, observe that a bounce is a discrete occurrence that will cause a discontinuity in the behaviour of the ball. Thus a bounce is an event occurrence, and a bounce detector is a function mapping *Behaviour Ball* to *Event Ball* (the value of the event occurrence being the configuration of the ball at the moment of impact):

```
detectBounce : Behaviour Ball → Event Ball
detectBounce = when detectImpact
```

A *Behaviour* for a ball that bounces perfectly elastically is thus defined:

```
elasticBall : Ball → Behaviour Ball
elasticBall b = let beh = fallingBall b
                in untilB beh (detectBounce beh) (elasticBall ∘ negateVel)
```

Intuitively, this says that an elastic ball should behave as a falling ball until a bounce is detected. At which point, the ball should have its velocity negated, and then have its configuration used to initialise a new *elasticBall* behaviour.

A *Behaviour* for a ball that collides perfectly inelastically with the ground can be defined similarly:

```
inelasticBall : Ball → Behaviour Ball
inelasticBall b = let beh = fallingBall b
                  in untilB beh (detectBounce beh) (λ _ → constant (0, 0))
```

The definition of *elasticBall* is a good example of a situation where a *Behaviour* should not be started until it is switched-in. If *elasticBall* had been defined using *untilB′*, then the residual *Behaviour* (the recursive call to *elasticBall*) would start at the same time as the subordinate behaviour (*fallingBall*), with its initial configuration being that of the ball when it bounces. Imagine the ball first bounces after 5 seconds. The overall behaviour immediately after the bounce would then be that of a ball 5 seconds after such a bounce! To someone viewing an animation of the ball, it would appear to "jump" 5 seconds into the future at the moment of the bounce.

This is not to say that there are never situations when it is desirous to have *Behaviours* starting before they are switched-in though, as will be seen shortly.

Repositioning Balls

The final addition to this model is the capacity for a ball to be arbitrarily moved to a new position (and given a new velocity) by some external actor. The intuitive way to express this would seem to be as follows:

```
resetBall : Event Ball → (Ball → Behaviour Ball) → Ball → Behaviour Ball
resetBall ev f b = untilB (f b) ev (resetBall ev f)
```

The idea would be that the first argument (*ev*) is the *Event* that controls the resets, the second argument (*f*) is a function that generates a ball *Behaviour* given a ball configuration, and the third argument (*b*) is the initial ball configuration. Thus, *resetBall ev elasticBall b* would behave as *elasticBall b* until an event occurs in *ev*, at which point it recursively starts *resetBall* using the same *ev* but a new initial ball configuration.

However, this is not how *resetBall* behaves. When the first reset occurs, the *Event* is reset along with the ball *Behaviour*. Consequently, the first event occurrence will trigger the reset repeatedly, and any occurrences thereafter will be ignored. For example, if the first event occurs after 3 seconds, then the reset will be triggered every 3 seconds, regardless of any subsequent events. To overcome this, there needs to be a way of retaining the event signal when a structural switch occurs, rather than restarting it. This is addressed in the next section.

3.3.5 Retaining Signals

It may seem that using *untilB′* in place of *untilB* in the definition of *resetBall* would solve the problem. However, while it would lead to the correct behaviour from the event signal, it would produce an incorrect new ball Behaviour (as discussed for *elasticBall*). Thus both capabilities are needed. However, rather than also providing combinators of the *untilB′* variety, CFRP variants addressing this problem provide a family of *runningIn* primitives that allow *Behaviours* and *Events* to start running before they are switched-in. This is achieved by fixing the start time of the *Behaviour* or *Event* such that when it is switched-in its start time does not change. In effect, the *runningIn* primitives coerce *Behaviours* and *Events* from signal generators to signals, thus providing the programmer with both first-class signals and first-class signal generators. These signals (running *Behaviours* or *Events*) can then be used in the definitions of other *Behaviours* and *Events* that have not yet been switched-in.

There are four functions in the *runningIn* family, one for each possible pair combination of *Event* and *Behaviour*. First consider *runningInBB*, which starts a *Behaviour* inside a *Behaviour*:

```
runningInBB : Behaviour A → (Behaviour A → Behaviour B) → Behaviour B
runningInBB beh f ≈ λ t₀ → f (λ _ → beh t₀) t₀
```

The first argument (*beh*) is the *Behaviour* to start running. The second argument (f) is a function that uses this *Behaviour* (which is really a signal, despite the lack of type distinction) to define another *Behaviour*. The semantics say that *beh* can be used in the definition of the second *Behaviour*, but that whenever *beh* is switched in, the local start time is ignored and the start time of the *runningInBB* expression is used instead.

The *runningIn* primitive that is needed for the Bouncing-Ball example is *runningInEB*, which starts an *Event* inside a *Behaviour*:

```
runningInEB : Event A → (Event A → Behaviour B) → Behaviour B
runningInEB ev f ≈ λ t₀ → f (λ tₑ → dropWhile ((< tₑ) ∘ fst) ∘ ev t₀) t₀
```

The semantics are similar to *runningInBB*, except that *dropWhile* $((< t_e) \circ$ *fst*$)$ is applied to the running *Event*. This is because the meaning of an *Event* is all event occurrences between the start time and the sample time (whereas a *Behaviour* is only concerned with the sample time). While the *Event* should start running before it is switched-in, only events that occur after it is switched-in should be observable.

A combinator that replaces a *Behaviour* whenever an event occurs in a specified event signal can now be defined as follows:

```
replaceBeh : Event E → Behaviour A → (E → Behaviour A) → Behaviour A
replaceBeh ev beh f = runningInEB ev (λ rev → replaceBehAux rev beh)
  where
    replaceBehAux : Event E → Behaviour A → Behaviour A
    replaceBehAux rev beh′ = untilB beh′ rev (λ e → replaceBehAux (notYet rev) (f e))
```

Initially, *replaceBeh ev f beh* behaves as *beh*. Whenever an event occurs in *ev*, the switching function *f* is applied to the value of that event to produce a *Behaviour*. That *Behaviour* is then switched-in, and the old *Behaviour* switched-out. The use of *runningInEB* prevents *ev* from being restarted. Note that the use of *notYet* in the recursive call to *replaceBehAux* is crucial. Without it, the switched-in *untilB* combinator would immediately switch again, leading to an infinite chain of switching at that time point.

Returning to the Bouncing-Ball example, *resetBall* can now be correctly defined as follows:

```
resetBall : Event Ball → (Ball → Behaviour Ball) → Ball → Behaviour Ball
resetBall ev f b = replaceBeh ev f (f b)
```

3.4 Unary FRP

Section 3.2.2 contained a conceptual definition of signal functions that map a single signal to a single signal. FRP models that take such signal functions as the central reactive abstraction will be referred to as Unary FRP (UFRP). The Yampa implementation [92] is based on UFRP. This section introduces a basic UFRP language, and then gives some examples of UFRP programming.

3.4.1 Signals, Signal Functions, and Events

The conceptual model of UFRP is based directly on the signals and signal functions introduced in Section 3.2:

$$\begin{aligned}&\mathit{Signal} : \mathit{Set} \rightarrow \mathit{Set}\\ &\mathit{Signal}\ A \approx \mathit{Time} \rightarrow A\\ &\mathit{SF} : \mathit{Set} \rightarrow \mathit{Set} \rightarrow \mathit{Set}\\ &\mathit{SF}\ A\ B \approx \mathit{Signal}\ A \rightarrow \mathit{Signal}\ B\end{aligned}$$

Signal functions are abstract first-class entities in the UFRP language. Signals, on the other hand, are second-class: they exist only indirectly through the signal functions.

However, the UFRP model does not lend itself well to having an additional signal type for discrete-time signals, as then signal-function variants for each possible input/output combination of continuous-time and discrete-time signals would be needed. Instead, UFRP embeds discrete-time signals within continuous-time signals. This is achieved by an abstract *Event* type, which is conceptually a (time-ordered and finite) list of event occurrences:

$$\begin{aligned}&\mathit{Event} : \mathit{Set} \rightarrow \mathit{Set}\\ &\mathit{Event}\ A \approx \mathit{List}\ (\mathit{Time} \times A)\end{aligned}$$

A discrete-time signal carrying elements of type A is then represented by a signal of type *Signal* (*Event* A). The (conceptual) value of such a signal at a point in time is the list of event occurrences up to that time point.

In Yampa, the *Event* type is *implemented* as an abstract option type:

```
data Event (A : Set) : Set where
  noEvent :       Event A
  event   : A → Event A
```

Thus, a signal of type *Signal* (*Event* A) would have a value of noEvent whenever the discrete-time signal is not defined, and a value of event v whenever the discrete-time signal is defined with value v. This is unsuitable for use at the semantic level, as it would allow *dense* event signals: signals where events are always occurring over a non-zero interval. However, it is sufficient to define the primitives in Yampa's discretely sampled implementation (which only approximates the ideal semantics).

3.4.2 Primitive Signal Functions

In a similar manner to CFRP, UFRP provides lifting functions that lift values and functions to the reactive level:

$$\begin{aligned}&\mathit{constant} : B \rightarrow \mathit{SF}\ A\ B\\ &\mathit{constant}\ b \approx \lambda\, s\ t \rightarrow b\end{aligned}$$

$\mathit{lift} : (A \to B) \to \mathit{SF}\ A\ B$
$\mathit{lift}\ f \approx \lambda\ s \to f \circ s$

Note that *constant* is completely polymorphic in its input signal. As discussed in Section 3.2.5, this is a way of embedding what are really signal generators into a signal-function setting.

In CFRP, *notYet*, *integral* and *when* are *host-language* functions operating on *Events* and *Behaviours*. Here, they are *signal functions*:

$\mathit{notYet} : \mathit{SF}\ (\mathit{Event}\ A)\ (\mathit{Event}\ A)$
$\mathit{notYet} \approx \lambda\ s \to \mathit{dropWhile}\ ((\leqslant 0) \circ \mathit{fst}) \circ s$

$\mathit{integral} : \mathit{SF}\ \mathbb{R}\ \mathbb{R}$
$\mathit{integral} \approx \lambda\ s\ t_1 \to \int_0^{t_1} (s\ t)\ dt$

$\mathit{when} : (A \to \mathit{Bool}) \to \mathit{SF}\ A\ (\mathit{Event}\ A)$

The definition of *when* is again omitted as it is substantially more involved than the other definitions in this section.

3.4.3 Primitive Routing Combinators

Signal functions are abstract entities in UFRP, and thus (outside of the conceptual level) they cannot be applied or composed as host-language functions. Instead, UFRP languages provide a set of routing combinators that can be used to construct whatever network structure is desired. The UFRP variant defined here provides two routing combinators as primitives. The first combinator is the *sequential composition* (denoted $\ggg$) of two signal functions:

$\_\ggg\_ : \mathit{SF}\ A\ B \to \mathit{SF}\ B\ C \to \mathit{SF}\ A\ C$
$\mathit{sf}_1 \ggg \mathit{sf}_2 \approx \mathit{sf}_2 \circ \mathit{sf}_1$

The second combinator (denoted &&& and pronounced "fan-out") applies two signal functions to the same input in parallel:

$\_\&\&\&\_ : \mathit{SF}\ A\ B \to \mathit{SF}\ A\ C \to \mathit{SF}\ A\ (B \times C)$
$\mathit{sf}_1\ \&\&\&\ \mathit{sf}_2 \approx \lambda\ s\ t \to (\mathit{sf}_1\ s\ t, \mathit{sf}_2\ s\ t)$

Data-flow combinators such as these are often easiest to understand graphically: see Figure 3.1.

3.4.4 Switching Combinators

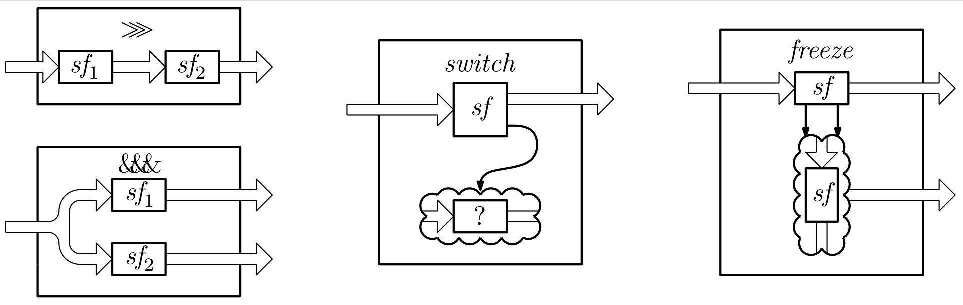
UFRP variants provide families of switching combinators that allow structural dynamism to be expressed. New first-class signal functions can be created and switched-in, and running signal functions can be switched-out. [92]

Here, only two primitive switching combinators are considered. The first is called *switch*:

$\mathit{switch} : \mathit{SF}\ A\ (B \times \mathit{Event}\ E) \to (E \to \mathit{SF}\ A\ B) \to \mathit{SF}\ A\ B$

This combinator takes two arguments, which are called the *subordinate signal function* and the *switching function*. Informally, the behaviour of *switch* is as follows. The subordinate signal function is applied to the input signal until the first occurrence in the output event signal. The output of the switching combinator is taken from the first component of the subordinate's output until this point. The switching function is then applied to the value of the event occurrence to produce a residual signal function. The residual signal function is then applied to the input signal, starting at the time of the event occurrence, and henceforth the output of the switching combinator is taken from the residual signal function.

Figure 3.1 UFRP primitive combinators



To express this formally, the following (semantic level) function is useful:

$$\begin{aligned}
&\mathit{advance} \;:\; \mathit{Time} \rightarrow \mathit{Signal}\ A \rightarrow \mathit{Signal}\ A\\
&\mathit{advance}\ d\ s\ t \;=\; s\ (t + d)
\end{aligned}$$

Intuitively, *advance* d s time-shifts the signal s forward in time by an amount d. This is needed because, unlike CFRP which had signal generators and start times, signal functions in UFRP live in their own *local* time frame.

Local Time: The time since a signal function was applied to its input signal.

Note that a signal function is applied to a signal either when the entire system starts, or when it is switched-in.

The *switch* combinator can now be defined as follows:

$$\begin{aligned}
&\mathit{switch} \;:\; \mathit{SF}\ A\ (B \times \mathit{Event}\ E) \rightarrow (E \rightarrow \mathit{SF}\ A\ B) \rightarrow \mathit{SF}\ A\ B\\
&\mathit{switch}\ \mathit{sf}\ f \;\approx\; \lambda\ s\ t \rightarrow \mathbf{let}\ (b, ev) \;=\; \mathit{sf}\ s\ t\\
&\qquad\qquad\qquad \mathbf{in\ case}\ \mathit{dropWhile}\ (\lambda\ (t_e, \_) \rightarrow t_e < 0)\ ev\ \mathbf{of}\\
&\qquad\qquad\qquad\quad [\,] \qquad\qquad \rightarrow b\\
&\qquad\qquad\qquad\quad (t_e, e) :: \_ \rightarrow (f\ e)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)
\end{aligned}$$

The key point is that the residual signal function (f e) only "starts" at the moment of switching (t_e). Thus, semantically, the input signal (s) has to be advanced by an amount t_e to shift it into the local time frame of the residual signal function. Consequently, the residual signal function only observes the input signal after the moment of switching (inclusive). The sampling time is then reduced ($t - t_e$) to shift the output signal back into the external time frame. The use of *dropWhile* serves the same purpose as in *runningInEB* in CFRP (Section 3.3.5): it hides any events that occurred before this signal function was switched-in.

Note that the value of the output signal at the moment of switching is taken from the residual signal function. An alternative design decision would be to have the output at that moment be taken from the subordinate signal function. UFRP provides a switching combinator called *dswitch* that has such behaviour:

$$\begin{aligned}
&\mathit{dswitch} \;:\; \mathit{SF}\ A\ (B \times \mathit{Event}\ E) \rightarrow (E \rightarrow \mathit{SF}\ A\ B) \rightarrow \mathit{SF}\ A\ B\\
&\mathit{dswitch}\ \mathit{sf}\ f \;\approx\; \lambda\ s\ t \rightarrow \mathbf{let}\ (b, ev) \;=\; \mathit{sf}\ s\ t\\
&\qquad\qquad\qquad \mathbf{in\ case}\ \mathit{dropWhile}\ (\lambda\ (t_e, \_) \rightarrow (t_e < 0) \vee (t_e \geqslant t))\ ev\ \mathbf{of}\\
&\qquad\qquad\qquad\quad [\,] \qquad\qquad \rightarrow b\\
&\qquad\qquad\qquad\quad (t_e, e) :: \_ \rightarrow (f\ e)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)
\end{aligned}$$

This is the same as the definition of *switch*, except that if the first event occurrence is *at* the sample time then it is discarded. Thus, at the sample time, the output is defined to be that of the subordinate signal function (b).

3.4.5 Freezing Signal Functions

UFRP also allows "running" signal functions to be "frozen" (transformed back into first-class entities, maintaining any accumulated internal state) [92]. These frozen signal functions can then be switched-in later using switching combinators. Here, only one freezing combinator is considered:

$$\mathit{freeze} : \mathit{SF}\ A\ B \to \mathit{SF}\ A\ (B \times \mathit{SF}\ A\ B)$$

Informally, *freeze* applies its subordinate signal function to the input signal to produce an output signal, but also emits a "frozen" copy of the (aged) subordinate signal function as an additional output. This frozen signal function is a first-class entity at the functional level, but is one that has already received some of its input.

An alternative way of looking at this is that *freeze* allows execution of a signal function to be suspended. Then, later, the frozen signal function can be resumed by switching it in. An example of this can be found in Section 7.4.

Defining this formally requires another (semantic level) function:

$$\begin{aligned}
&\mathit{splice} : \mathit{Signal}\ A \to \mathit{Signal}\ A \to \mathit{Time} \to \mathit{Signal}\ A \\
&\mathit{splice}\ s_1\ s_2\ t_x\ t \mid t < t_x = s_1\ t \\
&\qquad\qquad\qquad \mid t \geqslant t_x = s_2\ (t - t_x)
\end{aligned}$$

Intuitively, *splice* composes two signals temporally, ending the first signal, and starting the second, at the given time (t_x).

The *freeze* combinator can now be defined as follows:

$$\begin{aligned}
&\mathit{freeze} : \mathit{SF}\ A\ B \to \mathit{SF}\ A\ (B \times \mathit{SF}\ A\ B) \\
&\mathit{freeze}\ \mathit{sf} \approx \lambda\ s\ t \to (\mathit{sf}\ s\ t, \lambda\ s' \to \mathit{advance}\ t\ (\mathit{sf}\ (\mathit{splice}\ s\ s'\ t)))
\end{aligned}$$

Note that the frozen signal function will only have processed input *strictly before* the time point at which it is frozen.

3.4.6 Example UFRP Programming

This section demonstrates UFRP programming by defining some library signal functions and combinators. Note that this is no longer at the conceptual level: thus signal functions are now abstract, and signals are not first class.

Initialised Integration

An initialised integral is defined similarly to CFRP:

$$\begin{aligned}
&\mathit{iIntegral} : \mathbb{R} \to \mathit{SF}\ \mathbb{R}\ \mathbb{R} \\
&\mathit{iIntegral}\ x = \mathit{integral} \ggg \mathit{lift}\ (+\ x)
\end{aligned}$$

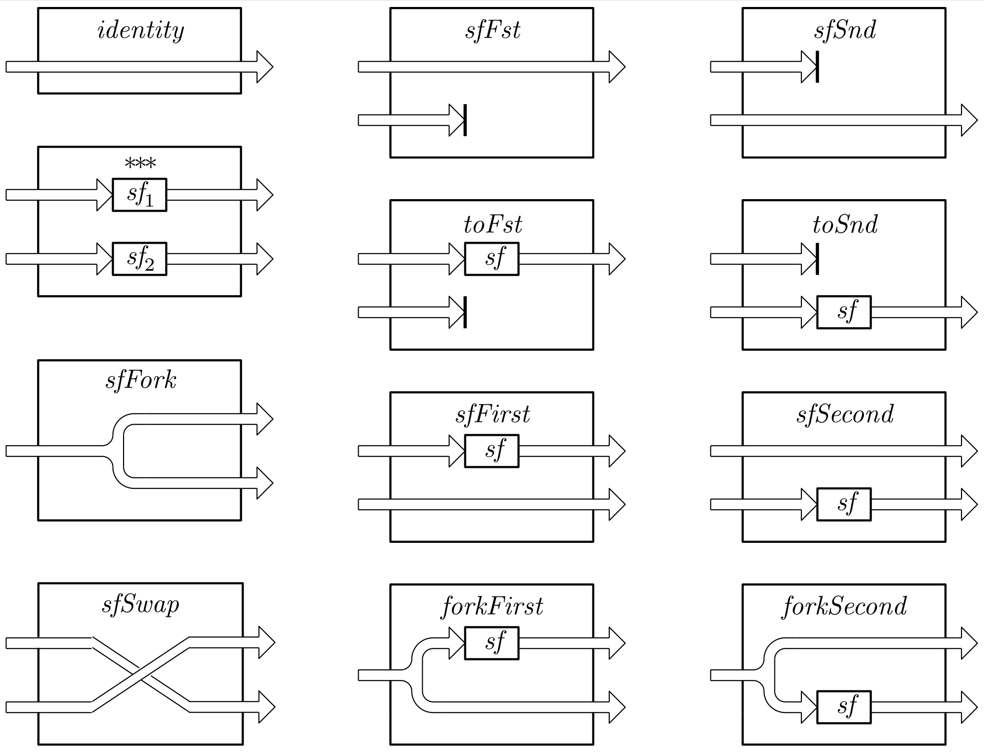
Routing Combinators

As previously mentioned, UFRP provides a set of routing combinators to compose signal functions. This set of combinators can be defined using *lift*, $\ggg$ and &&&:

$$\begin{aligned}
&\mathit{identity} : \mathit{SF}\ A\ A \\
&\mathit{identity} = \mathit{lift}\ \mathit{id}
\end{aligned}$$

$$\begin{aligned}
&\mathit{sfFst} : \mathit{SF}\ (A \times B)\ A \\
&\mathit{sfFst} = \mathit{lift}\ \mathit{fst}
\end{aligned}$$

Figure 3.2 Additional UFRP routing combinators



$sfSnd\ :\ SF\ (A \times B)\ B$
$sfSnd\ =\ lift\ snd$

$sfSwap\ :\ SF\ (A \times B)\ (B \times A)$
$sfSwap\ =\ lift\ swap$

$sfFork\ :\ SF\ A\ (A \times A)$
$sfFork\ =\ lift\ fork$

$toFst\ :\ SF\ A\ C \to SF\ (A \times B)\ C$
$toFst\ sf\ =\ sfFst \ggg sf$

$toSnd\ :\ SF\ B\ C \to SF\ (A \times B)\ C$
$toSnd\ sf\ =\ sfSnd \ggg sf$

$\_***\_\ :\ SF\ A\ C \to SF\ B\ D \to SF\ (A \times B)\ (C \times D)$
$sf_1 *** sf_2\ =\ toFst\ sf_1\ \&\&\&\ toSnd\ sf_2$

$sfFirst\ :\ SF\ A\ B \to SF\ (A \times C)\ (B \times C)$
$sfFirst\ sf\ =\ sf *** identity$

$sfSecond\ :\ SF\ B\ C \to SF\ (A \times B)\ (A \times C)$
$sfSecond\ sf\ =\ identity *** sf$

$forkFirst\ :\ SF\ A\ B \to SF\ A\ (B \times A)$
$forkFirst\ sf\ =\ sfFork \ggg sfFirst\ sf$

$forkSecond\ :\ SF\ A\ B \to SF\ A\ (A \times B)$
$forkSecond\ sf\ =\ sfFork \ggg sfSecond\ sf$

These combinators are best understood graphically: see Figure 3.2.

Additional Switching Combinators

The following switching combinator will be useful for the Bouncing-Ball example:

$$\begin{array}{l} \mathit{switchWhen} \,:\, \mathit{SF}\ A\ B \to \mathit{SF}\ B\ (\mathit{Event}\ E) \to (E \to \mathit{SF}\ A\ B) \to \mathit{SF}\ A\ B \\ \mathit{switchWhen}\ \mathit{sf}\ \mathit{sfe} \ = \ \mathit{switch}\ (\mathit{sf} \ggg \mathit{forkSecond}\ \mathit{sfe}) \end{array}$$

Essentially, *switchWhen* is *switch* specialised to the case where the event signal that controls the switch only depends on the output of the subordinate signal function. It differs from *switch* in that the subordinate signal function has been split into two: one to produce the output and one to produce the event.

Recall from Section 3.3.5 that it is sometimes necessary to use *notYet* when defining a switching combinator recursively, to prevent an infinite chain of switching. Note that this was not required in the definition of *elasticBall*, despite the recursive call, because the *when* primitive never produces an event at the moment it is switched-in. As can be imagined, this can be a subtle source of bugs in FRP programs. To address this, UFRP provides a recursive switching combinator that incorporates the *notYet* primitive in its definition:

$$\begin{array}{l} \mathit{rswitch} \,:\, \mathit{SF}\ A\ (B \times \mathit{Event}\ E) \to (E \to \mathit{SF}\ A\ (B \times \mathit{Event}\ E)) \to \mathit{SF}\ A\ B \\ \mathit{rswitch}\ \mathit{sf}\ f \ = \ \mathit{switch}\ \mathit{sf}\ (\lambda\ e \to \mathit{rswitch}\ (f\ e \ggg \mathit{sfSecond}\ \mathit{notYet})\ f) \end{array}$$

The switching function of *rswitch* produces a new subordinate signal function that replaces the existing subordinate signal function. This differs from *switch*, where the switching function produces a residual signal function that replaces the *entire* switching combinator. The intent is that any recursive switching should be defined using *rswitch*, rather than by using *switch* and host-language recursion.

A recursive variant of *switchWhen* is also useful. However, there are several possible meanings for an *rswitchWhen* combinator: Whenever a structural switch occurs, the signal function that generates the event could either restart, continue running, or be replaced by a newly computed signal function. Here, the first option is chosen:

$$\begin{array}{l} \mathit{rswitchWhen} \,:\, \mathit{SF}\ A\ B \to \mathit{SF}\ B\ (\mathit{Event}\ E) \to (E \to \mathit{SF}\ A\ B) \to \mathit{SF}\ A\ B \\ \mathit{rswitchWhen}\ \mathit{sf}\ \mathit{sfe}\ f \ = \ \mathit{rswitch}\ (\mathit{sf} \ggg \mathit{forkSecond}\ \mathit{sfe})\ (\lambda\ e \to f\ e \ggg \mathit{forkSecond}\ \mathit{sfe}) \end{array}$$

Finally, an equivalent of the *replaceBeh* combinator (Section 3.3.5) can be defined as follows:

$$\begin{array}{l} \mathit{replace} \,:\, \mathit{SF}\ A\ B \to (E \to \mathit{SF}\ A\ B) \to \mathit{SF}\ (A \times \mathit{Event}\ E)\ B \\ \mathit{replace}\ \mathit{sf}\ f \ = \ \mathit{rswitch}\ (\mathit{sfFirst}\ \mathit{sf})\ (\lambda\ e \to \mathit{sfFirst}\ (f\ \ e)) \end{array}$$

Note that the use of *rswitch* means that there is no need to use *notYet* in this definition.

Bouncing-Balls Revisited

The section concludes by adapting the Bouncing-Ball example from Section 3.3.4 to the setting of UFRP.

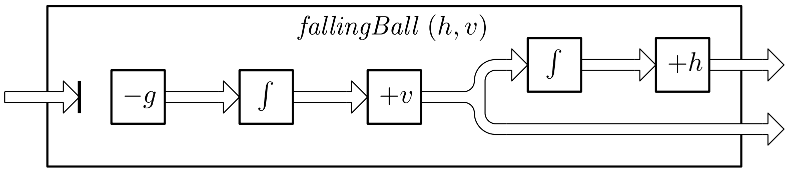
First, *fallingBall* is re-defined as a signal function:

$$\begin{array}{l} \mathit{fallingBall} \,:\, \mathit{Ball} \to \mathit{SF}\ A\ \mathit{Ball} \\ \mathit{fallingBall}\ (h, v) \ = \ \mathit{constant}\ (-g) \ggg \mathit{iIntegral}\ v \ggg \mathit{forkFirst}\ (\mathit{iIntegral}\ h) \end{array}$$

This code is less clear than its CFRP equivalent. The graphical representation in Figure 3.3 may be helpful.

Programming with routing combinators is often more awkward than just applying functions to arguments, and in practice some convenient syntax is usually provided by an implementation

Figure 3.3 A signal function network modelling a falling ball



to alleviate the burden. For example, Yampa, which is structured using *Arrows* [59], makes use of Paterson's arrow notation [101]. The advantage of such notation is that it allows *pointwise* programming; that is, intermediate signals can be named. In this notational style, *fallingBall* could be defined as follows:

$$\begin{aligned}
&\mathit{fallingBall} : \mathit{Ball} \rightarrow \mathit{SF}\ A\ \mathit{Ball}\\
&\mathit{fallingBall}\ (h_0, v_0) = \mathbf{proc}\ \_ \rightarrow \mathbf{do}\\
&\qquad v \leftarrow \mathit{iIntegral}\ v_0 \prec -g\\
&\qquad h \leftarrow \mathit{iIntegral}\ h_0 \prec v\\
&\qquad \mathit{identity} \quad\ \prec (h, v)
\end{aligned}$$

The basic idea is that input signals are placed on the right, signal functions appear in the middle, and output signals are bound to identifiers on the left. The overall input signal appears after **proc** (in this case an underscore, as it is not used), and the overall output is that produced by the signal function on the final line. The signal function definitions may not depend on the signals. For a more detailed explanation of the notation in the context of FRP, consult Nilsson et al. [92].

The remaining ball definitions are straightforward:

$$\begin{aligned}
&\mathit{detectBounce} : \mathit{SF}\ \mathit{Ball}\ (\mathit{Event}\ \mathit{Ball})\\
&\mathit{detectBounce} = \mathit{when}\ \mathit{detectImpact}\\
&\mathit{elasticBall} : \mathit{Ball} \rightarrow \mathit{SF}\ A\ \mathit{Ball}\\
&\mathit{elasticBall}\ b = \mathit{rswitchWhen}\ (\mathit{fallingBall}\ b)\ \mathit{detectBounce}\ (\mathit{fallingBall} \circ \mathit{negateVel})\\
&\mathit{inelasticBall} : \mathit{Ball} \rightarrow \mathit{SF}\ A\ \mathit{Ball}\\
&\mathit{inelasticBall}\ b = \mathit{switchWhen}\ (\mathit{fallingBall}\ b)\ \mathit{detectBounce}\ (\lambda\ \_ \rightarrow \mathit{constant}\ (0,0))\\
&\mathit{resetBall} : (\mathit{Ball} \rightarrow \mathit{SF}\ A\ \mathit{Ball}) \rightarrow \mathit{Ball} \rightarrow \mathit{SF}\ (A \times \mathit{Event}\ \mathit{Ball})\ \mathit{Ball}\\
&\mathit{resetBall}\ f\ b = \mathit{replace}\ (f\ b)\ f
\end{aligned}$$

3.4.7 Single-Kinded Signals

The UFRP model defined in Section 3.4.1 represents discrete-time signals by embedding an abstract *Event* type in continuous-time signals. This approach is called *single-kinded* UFRP, as there is really only one signal kind. This uniform treatment of continuous-time and discrete-time signals fits well with the idea of signal functions being the core concept and there only being one kind of signal function. However, making *Event* first class allows a "mischievous" programmer to violate the conceptual model of events, such as by defining dense event occurrences (an infinite number of occurrences over a finite time interval). Consequently, an implementation cannot safely carry out optimisations that are predicated on events occurring non-densely, even though that is the intent. The drawbacks of single-kindedness are discussed further in Section 4.1.1.

3.5 Advantages of a Signal-Function Abstraction

Of the two FRP variants introduced thus far, CFRP is based around first-class signal generators, whereas UFRP is based around a first-class signal-function abstraction. This thesis develops and studies a UFRP-inspired FRP variant called N-ary FRP, where signal functions are the primary notion and signals are secondary. To motivate this design choice (as opposed to a more CFRP-like language), this section contains a brief discussion of some of the advantages that this approach offers.

3.5.1 Implementation Implications

Implementing first-class signals efficiently in their full generality is challenging [36, 37, 63]. The essential difficulty is that signals are *time-varying* entities occurring at the functional level where everything notionally must be *time-invariant* so as not to break referential transparency. The key to solving this apparent contradiction is to adopt the view that the signal abstraction represents the *entire* signal, which is time invariant. However, if signals are truly first-class, then they can be put into data structures or be part of closures, and be kept there for a long time without any connection to the outside world. Yet if space and time leaks are to be avoided, signals have to be *implemented* as truly time-varying values by updating them as soon as there is a change [36, 100].

To my knowledge, all practically useful FRP implementations supporting first-class signals resort to imperative techniques to address this. For example, *runningIn* was implemented by updating the running *Behaviour* or *Event* as a side effect (using Haskell's *unsafePerformIO*) of consuming the produced signal (that need not depend on the running *Behaviour* or *Event* at all points of time; in fact, normally would not). For another example, the latest version of Elerea (Version 2) maintains a pool of (weak) references to all active stateful signal computations to enable all of them to be updated, regardless of whether or not the result of an individual computation is currently being used, by making a sweep over the pool at every time step [100]. On the other hand, Version 1 of Elerea avoids the problem by simply not updating any signals that are not contributing to output at the current execution step, but this has the disadvantage of breaking referential transparency [99].

In contrast, an approach based on signal functions can be implemented remarkably simply and purely functionally [92]. In essence, a signal function is just a state transition function taking an input sample and current state to an output sample and new state. As the composition of such state transition functions is another state transition function, the entire system just becomes a state transition function. Signal functions, like signal generators, are *time-invariant*, so giving them first-class status at the functional level is trivial.

Another issue concerns sharing. As signal generators essentially are functions mapping a start time to a signal, the normal lazy evaluation machinery of a language like Haskell is not enough to ensure that signals generated by the same generator applied to the same start time are shared. This leads to a lot of redundant computation unless addressed, in particular for recursively defined signal generators. The usual solution is to employ some form of memoisation (again using imperative techniques) [36]. The memoisation is usually done internally, hidden by the abstractions; although Version 2 of the Elerea implementation provides an explicit memoisation primitive as memoising everything is usually redundant and has a negative impact on performance [100]. In contrast, with signal functions it is easy to arrange that each signal sample is computed exactly once and distributed to where it is needed, thus avoiding any risk of lost sharing.

Of course, what matters to an end user is not the complexity of an implementation, but the facilities provided, how easy they are to use, and the quality of the performance. As to the comparative performance of FRP implementations based on signals or signal functions, there is not yet a simple answer. Lots of research, implementation, and practical evaluation is still needed.

However, note that Yampa, despite having scalability issues (see Section 3.5.2), has proved to be quite efficient for many applications as witnessed by video-game implementations [20, 27] and the Yampa synthesiser [43]. It seems likely that this is mainly due to the implementation being purely functional, and functional compilers being good at compiling purely functional code. Moreover, the work on *Causal Commutative Arrows* [75, 77] has shown that static signal function networks can be executed very efficiently.

3.5.2 Routing

An FRP program defines a synchronous data-flow network. The nodes of this network are signal functions, regardless of whether signals or signal functions are first-class abstractions in the language used to define the network. In languages with first-class signals, the routing of the network is defined at the functional level by host-language functions, hiding it from the reactive level. On the other hand, a language with a first-class signal-function abstraction can construct the network using routing combinators that operate on signal functions. This allows *all* routing to be defined at the reactive level, giving much greater scope for optimisation than when the routing is hidden in the host language. However, the UFRP model, despite having first-class signal functions, is still insufficient.

In UFRP, signal functions have only a single input and a single output. Consequently, the only way to represent signal functions operating on (or returning) more than one signal is to exploit the fact that a product of signals is (in this model) isomorphic to a single signal carrying a product of elements of the constituent signals. For example, a signal function that maps a pair of signals carrying integers to another pair of signals carrying integers has type:

$SF\ (\mathbb{Z}, \mathbb{Z})\ (\mathbb{Z}, \mathbb{Z})$

This means that there is no distinction (and cannot be) between a signal that carries a pair of values, and one that is the result of pairing two independent signals.

Moreover, exploiting this isomorphism is often the *only* way to route signals between signal functions. Signals are grouped together into a single signal according to the structure of signal function composition, and then, at the functional level, values of this signal are regrouped so as to enable decomposition according to the structure of the receiving signal function. This approach hides the routing from the reactive level, and creates artificial interdependencies between independent signals. This makes it difficult to implement the UFRP model in a way that scales well, such as through direct point-to-point communication between signal functions or minimisation of redundant computation through change propagation (as proposed in Chapter 8).

The UFRP model certainly does not rule out all optimisation opportunities, as evidenced by the latest Yampa implementation [90]. However, overcoming these limitations in a more comprehensive and systematic way necessitates internalising the routing at the reactive level, as well as introducing n-ary signal functions that truly map multiple *independent* input signals to multiple *independent* output signals. It is for these reasons that the N-ary FRP model has been developed (Chapter 4), which is based around such n-ary signal functions.

3.5.3 Switching

Recall the *replace* combinator from UFRP (Section 3.4.6). Unlike the *replaceBeh* combinator from CFRP (Section 3.3.5), its definition makes clear which signals are restarted at the moment of switching, and which are maintained. Primarily, this is because of the modular nature of signals functions: they are parametrised on their input, which is explicitly received from some external source. For switching combinators, this makes a clear distinction between internal signals (produced by subordinate or residual signal functions), which exist in their local time frame, and external signals (the input), which exist in an external time frame and are unaffected by the structural switch. In CFRP, this was not possible without using the *runningIn* primitive to coerce signal generators into signals.

As well as being more complicated, the *runningIn* primitive also gives rise to a number of theoretical and practical problems. In particular, it leads to confusion between signals and signal generators (which are both typed as *Behaviours*), and can easily lead to ill-defined programs when writing recursive *Behaviours*. A detailed discussion of these issues can be found in Courtney [25].

3.5.4 Signal Function Objects

By making signal functions a first-class abstraction, an FRP implementer has great freedom in choosing their representation and, subsequently, in exploiting information manifest in this representation. For example, Yampa encodes simple properties about signal functions in their representation, which in favourable circumstances allows compositions of signal functions to be fused for better performance [90]. One of the goals of the present work is to identify properties of signal functions that could enable such optimisation in a more systematic and formally justifiable manner (Chapter 8).

Similarly, as will be discussed in chapters 7 and 9, being able to associate additional information with signal functions *at the type level* allows certain safety guarantees, such as the absence of instantaneous feedback loops, to be enforced statically. If signal functions were ordinary host-language functions on signals, then it would not be possible to take such information into account if it truly relates to the function as opposed to its argument or result.

As described in Section 3.4.5, UFRP allows a switched-in signal function to be "frozen"; that is, switched-out of the network and returned to a first-class entity at the functional level, maintaining the internal state it had at the moment it was switched-out. At some later point, the frozen signal function can be switched-in again. This is a powerful capability, forming the basis of Yampa's collection-based switching primitives that allow highly dynamic signal function networks to be described. The same fundamental mechanism is also used in the virtual-reality project FRVR [12] where, through a Yampa extension, it is used to implement an editor undo facility by capturing the system state as frozen signal functions at various points in time. This allows interaction to resume from any saved point at a later stage, thereby undoing the effects of any intervening interaction (see Section 7.4 for an example of this). It would seem hard to replicate the freezing functionality in a setting with first-class signals.

3.5.5 Other Applications

Signal functions also have applications beyond FRP, making them interesting to study in their own right. The connections to the synchronous data-flow languages, and to modelling and simulation languages such as Simulink, were mentioned in Section 1.1. Functional Hybrid Modelling (FHM) [93] is an approach to modelling and simulation, in part inspired by FRP, where signal functions are generalised to relations on signals. For efficient simulation, while still allowing dynamism, these relations are compiled to native simulation code using the LLVM just-in-time compiler [44]. As the notions of signal relations and signal functions are related, and as it would be desirable to have signal functions in the FHM setting, the work in this thesis is potentially of use for FHM. Conversely, FHM's just-in-time compilation strategy could be applied in FRP implementations.

3.6 Conclusions

The notion of a signal is central to any FRP instance. As discussed, it is crucial to be able to start the computation of a signal at any desired point in time in order to support dynamism, both for reasons of expressivity and to avoid space and time leaks. This suggests a notion of signal generators as the central first-class abstraction. But first-class generators alone are not enough: the ability to refer to existing signals from within the definition of a generator is needed as well, suggesting that signals too should be first-class entities. One approach to overcoming this is the *runnningIn* primitive of CFRP, even though a signal in that particular formulation ends up being disguised as a *Behaviour* or *Event*; that is, as a signal *generator*. As a more recent example, Elerea also provides both signals and signal generators as first-class abstractions, but this time carefully distinguished at the type level [99, 100]. Either way, once signals are first-class entities, signal functions come for free.

However, as seen with Unary FRP, an alternative is to make *signal functions* the central first-class abstraction. They then play the role of generators, as a signal will be generated whenever a signal function is applied to a signal, either when the system first starts or when a new signal function is switched-in. Furthermore, signal functions are parametrised on their input, allowing residual signal functions to receive already existing signals. Thus neither signals nor signal generators need to be first-class.

So, is it better to have first-class signals (and generators) or first-class signal functions? There are pros and cons to each, many related to the specifics of a particular setting (embedded or stand-alone implementation, the facilities of the host language if an embedded approach is chosen, intended application area, etc.), and some somewhat subjective. Moreover, they are not mutually exclusive; for example, Grapefruit provides first-class signal and signal-function abstractions [63]. Yet, as discussed in Section 3.5, making signal functions first class has many advantages—in particular, it allows allows for a stricter separation between the functional and reactive layers. However, this is not to say that CFRP-like approaches are not viable; recent FRP implementations [23, 37, 63, 100] have shown that they are.

Chapter 4

N-ary FRP

This chapter describes a new FRP language called N-ary FRP. Section 4.1 defines the underlying conceptual model; Section 4.2 defines the primitives of the language in terms of that conceptual model; Section 4.3 gives some examples of N-ary FRP programming; and Section 4.4 contains a discussion about the totality of the language.

4.1 N-ary FRP Conceptual Model

As discussed in Section 3.5.2, the single-kinded UFRP model, while both simple and expressive, has a number of inherent problems, practical as well as conceptual. This section discusses some further issues with this model, and then introduces a refined conceptual model based on multi-kinded n-ary signal functions. This model, which underlies N-ary FRP, will serve as the foundation for the rest of this thesis.

4.1.1 Multi-Kinded Signals

As discussed in Section 3.2.3, many versions of FRP cater for the implementation of hybrid systems by supporting multi-kinded signals. On the other hand, in single-kinded FRP, discrete-time signals are defined in terms of continuous-time signals. As discussed in Section 3.4.7, this does not respect the conceptual model of events, and can lead to semantic infelicities.

Another problem of single-kinded signals is that some operations need to be done differently on the two kinds of signal in order to maintain central properties of the signal kind in question. For example, in a typical sampled implementation, it may be necessary to insert or delete samples of continuous-time signals to mediate between different sampling rates. However, for event signals, duplicating or eliminating event occurrences would often be disastrous. There may be specific versions of such operations that work correctly for events, but as any operation that works on polymorphic signals is also applicable to event signals, there is nothing to enforce that these specific operations are used in place of the generic ones. An example of this issue is discussed in Section 5.4.

Furthermore, many continuous-time signals are piecewise constant (mainly because of interaction with event signals). However, if all signals are continuous-time signals, without any further guaranteed properties, then there is not much that can be gained from this observation.

This is all in sharp contrast to multi-kinded FRP that makes a strict distinction between continuous-time and discrete-time signals, allowing the differences to be used for both gaining semantic precision and better implementation.

Consequently, it is desirable to make a clear type-level distinction between different kinds of signal. To this end, N-ary FRP identifies three distinct signal kinds:

- *Continuous Signal*: A general continuous-time signal that is always defined.
- *Event Signal*: A discrete-time signal only defined at an at-most-countable set of points in time. Each time point at which an event signal is defined is known as an *event occurrence*.
- *Step Signal*: A continuous-time piecewise-constant signal that is always defined. Its value only changes at an at-most-countable set of points in time.

4.1.2 N-ary Signal Functions

UFRP signal functions have only a single input and single output. As discussed in Section 3.5.2, this approach hides the network routing from the reactive level, and creates artificial interdependencies between independent signals. This limits the implementation techniques and optimisations that can be applied.

To address these routing limitations, and to cater for multi-kinded signals, n-ary signal functions are introduced: signal functions that can have more than one input or output. These n-ary signal functions are defined on *signal vectors*, conceptually products of heterogeneous signals, rather than signals.

The crucial point is that the different kinds of signal, and vectors of such signals, are defined only as an integral part of the signal-function abstraction. In this model, signals (and signal vectors) are second class and completely internalised at the reactive level. Thus there cannot be signals of signals, nor signals of signal vectors. This means that the N-ary FRP implementer has great freedom in choosing the representations of signals, signal functions, and the routing between them; and in exploiting those choices.

4.1.3 Signal Vector Descriptors

First, an auxiliary notion is required. A *signal vector descriptor* is a type-level value that describes key characteristics of a signal vector. Signal vector descriptors only exist at the type level of the N-ary FRP language, and are only used to index signal-function types.

The characteristics of interest are the *kind* of the signal, and the *type* of the values carried by the signal. Thus one descriptor is introduced for each signal kind, each parametrised on the value type, and a pairing descriptor to construct vectors of more than one signal:

```
data SVDesc : Set where
  C   : Set                → SVDesc   -- Continuous signal
  E   : Set                → SVDesc   -- Event signal
  S   : Set                → SVDesc   -- Step signal
  _,_ : SVDesc → SVDesc → SVDesc   -- product of signals
```

4.1.4 Refined Signals and Signal Functions

The conceptual definition of signals is now refined as follows:

- Continuous signals remain functions from time to value, as before.
- Event signals are modelled as an optional initial event and a function from time to a finite list of event occurrences. These occurrences are represented as pairs of a (strictly positive) time delta and a value.
- Step signals are modelled as an initial value and a function from time to a finite list of changes. These changes are represented as pairs of a (strictly positive) time delta and a value.

The time-delta–value pairs will be referred to as *occurrences*. Lists of occurrences will be referred to as *change lists*, and functions mapping time to change lists will be referred to as *change prefixes*. Also, the type $Time^+$ will denote the set of strictly positive time, with Δt a synonym for use when a (strictly positive) *time delta* is intended:

$$
\begin{aligned}
&\mathit{Time}^+ : \mathit{Set}\\
&\mathit{Time}^+ \approx \{t \in \mathbb{R} \mid t > 0\}\\
&\Delta t : \mathit{Set}\\
&\Delta t = \mathit{Time}^+\\
&\mathit{ChangeList} : \mathit{Set} \to \mathit{Set}\\
&\mathit{ChangeList}\ A = \mathit{List}\ (\Delta t \times A)\\
&\mathit{ChangePrefix} : \mathit{Set} \to \mathit{Set}\\
&\mathit{ChangePrefix}\ A = \mathit{Time} \to \mathit{ChangeList}\ A
\end{aligned}
$$

Signal vectors are thus defined:

$$
\begin{aligned}
&\mathit{SigVec} : \mathit{SVDesc} \to \mathit{Set}\\
&\mathit{SigVec}\ (\mathsf{C}\ A) &&= \mathit{Time} \to A\\
&\mathit{SigVec}\ (\mathsf{E}\ A) &&= \mathit{Maybe}\ A \times \mathit{ChangePrefix}\ A\\
&\mathit{SigVec}\ (\mathsf{S}\ A) &&= A \times \mathit{ChangePrefix}\ A\\
&\mathit{SigVec}\ (as, bs) &&= \mathit{SigVec}\ as \times \mathit{SigVec}\ bs
\end{aligned}
$$

Finally, signal functions are refined to operate on signal vectors:

$$
\begin{aligned}
&\mathit{SF} : \mathit{SVDesc} \to \mathit{SVDesc} \to \mathit{Set}\\
&\mathit{SF}\ as\ bs \approx \mathit{SigVec}\ as \to \mathit{SigVec}\ bs
\end{aligned}
$$

4.1.5 Why Change Prefixes?

The change-prefix representation is chosen because it ensures that the model is causal, and that occurrences are countable and not simultaneous.

To ensure that the model is causal, a change prefix maps a time to a finite list of occurrences, *up to that point in time*. Crucially, this means that at any time point, the times and values of all occurrences *up to that time point* can be computed without knowing the times of future occurrences. If just a change list was used to represent Event and Step signals, then computing all occurrences *up to* a time point would require knowing the time of the first occurrence *after* that time point, because only after that occurrence time had been compared with the current time could it be determined that it had not yet occurred.

As change lists are required to be finite, the change prefix representation also ensures that the number of occurrences is at most countable: there may be countably infinitely many in the limit as time tends towards infinity, but only a finite number up to any specific point.

Using *strictly positive* time deltas ensures that there cannot be several occurrences simultaneously. However, a consequence of this is that a change list cannot represent an occurrence at the first point in time (referred to henceforth as $time_0$). A Step signal is thus a change prefix paired with an initial value, while an Event signal is a change prefix paired with an optional initial event occurrence.

However, some additional constraints are required that the change prefix definition does not enforce:

Stable: The change list produced by a change prefix at time t must be the same as the prefix up to t of all change lists produced by the same change prefix at any time after t.

Non-Divining: The change list produced at any sample time must not extend beyond that sample time.

Intuitively, *stable* means that "history must not be re-written", and *non-divining* means that it must not be possible to "see into the future". A change prefix is said to be *coherent* if it satisfies both constraints. This could be incorporated into the change-prefix data structure, but that would substantially complicate the definitions in this thesis. Thus, it is instead stated as a side condition that is required to hold for all change prefixes in the model:

$$\begin{array}{l} \mathit{Coherent} \;:\; \mathit{ChangePrefix}\ A \rightarrow \mathit{Set} \\ \mathit{Coherent}\ \mathit{cp} \;=\; \forall\ t_1\ t_2 \rightarrow t_1 \leqslant t_2 \rightarrow \mathit{cp}\ t_1 \equiv \mathit{takeIncl}\ t_1\ (\mathit{cp}\ t_2) \end{array}$$

The *takeIncl* t function takes the greatest prefix of a change list such that the sum of its time deltas is at most t. Its definition can be found in Appendix B.1. When t_2 is greater than t_1 this ensures that the change prefix is *stable*; when t_1 and t_2 are equal this ensures that the change prefix is *non-divining*.

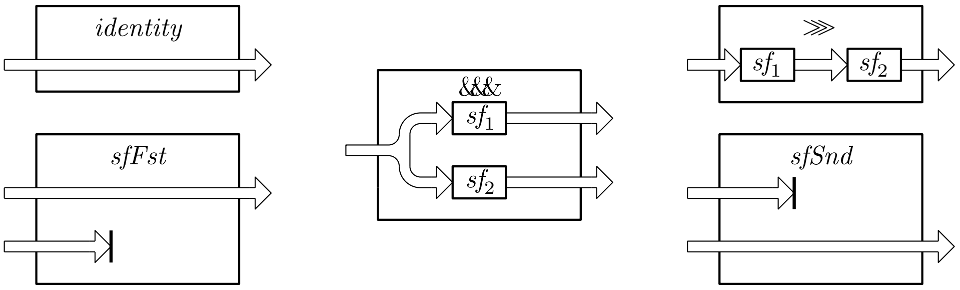
4.2 N-ary FRP Primitives

The primitives of an FRP language can be divided into:

- *routing primitives*, which express static network structure;
- *dynamic combinators*, which express dynamic network structure;
- *lifting functions*, which lift functions from the functional level to the reactive level by pointwise application (for stateless signal processing);
- *primitive signal functions*, which perform stateful signal processing.

This section introduces the primitives of N-ary FRP, giving their semantics in terms of the conceptual model from Section 4.1. When the semantics of a signal function is particularly verbose, only its type is given and its definition is relegated to Appendix B.

Figure 4.1 N-ary FRP routing primitives



4.2.1 Static Routers

As discussed in Section 3.5.2, a goal of N-ary FRP is to express all routing at the reactive level. To this end, there is a set of five (Arrows [59] inspired) primitives that exist purely for routing purposes (see Figure 4.1). All routing should be expressed using these primitives (as opposed to lifting routing functions from the functional level) so that an implementation can fully exploit this information.

Remember: an *implementation* of N-ary FRP is not required to be structured in a way that corresponds *directly* to the conceptual definitions below. All that is required is that the *semantics* of the implemented routing corresponds to the conceptual model.

The routing primitives are further subdivided into three *atomic routers* (which are signal functions) and two *routing combinators* (which are signal-function combinators). The atomic routers are defined as follows:

$identity : SF\ as\ as$
$identity \approx id$

$sfFst : SF\ (as, bs)\ as$
$sfFst \approx fst$

$sfSnd : SF\ (as, bs)\ bs$
$sfSnd \approx snd$

The routing combinators are taken from UFRP (see Section 3.4.3):

$\_ \ggg \_ : SF\ as\ bs \to SF\ bs\ cs \to SF\ as\ cs$
$sf_1 \ggg sf_2 \approx sf_2 \circ sf_1$

$\_\&\&\&\_ : SF\ as\ bs \to SF\ as\ cs \to SF\ as\ (bs, cs)$
$sf_1\ \&\&\&\ sf_2 \approx \lambda\ s \to (sf_1\ s, sf_2\ s)$

This set of primitives is minimal in the sense that any acyclic static network structure can be described by them, yet none of these primitives can be defined in terms of the other four. There are of course other sets of minimal combinators that can likewise express such routing. In Section 4.3.1 the expressiveness of these primitives is demonstrated by using them to define the set of UFRP routing combinators from Section 3.4.6.

To express cyclic routing (feedback), an additional routing combinator is required. Feedback is important facility in FRP (and synchronous data-flow generally). However, it is important not to introduce ill-defined feedback that could cause an implementation to loop at run-time. Ideally, the language should disallow ill-defined feedback without enforcing conservative restrictions on the well-defined feedback that is allowed. This is not a trivial concern, so introducing feedback combinators is postponed until Chapter 7. Until then, only acyclic networks will be considered.

4.2.2 Dynamic Combinators

The N-ary FRP language defined in this thesis contains two dynamic combinators. These are the *switch* and *freeze* combinators from UFRP (sections 3.4.4 and 3.4.5), but refined for the N-ary FRP model. It should also be possible to extend N-ary FRP with further dynamic combinators along the lines of Yampa's collection-based switches [92], but this remains the subject of future work.

$$\begin{aligned} &\mathit{switch} &&: \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A) \to (A \to \mathit{SF}\ \mathit{as}\ \mathit{bs}) \to \mathit{SF}\ \mathit{as}\ \mathit{bs} \\ &\mathit{freeze} &&: \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{C}\ (\mathit{SF}\ \mathit{as}\ \mathit{bs})) \end{aligned}$$

The formal definitions of these signal functions are somewhat more involved in the N-ary FRP model (see Appendix B.5), but in essence they are the same as in UFRP.

4.2.3 Lifting Functions

There is a family of lifting functions that allow pure functions to be lifted from the functional level to the reactive level in a pointwise fashion:

$$\begin{aligned} &\mathit{liftC} &&: (A \to B) \to \mathit{SF}\ (\mathsf{C}\ A)\ (\mathsf{C}\ B) \\ &\mathit{liftS} &&: (A \to B) \to \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{S}\ B) \\ &\mathit{liftE} &&: (A \to B) \to \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{E}\ B) \\ &\mathit{liftC2} &&: (A \to B \to Z) \to \mathit{SF}\ (\mathsf{C}\ A, \mathsf{C}\ B)\ (\mathsf{C}\ Z) \\ &\mathit{liftS2} &&: (A \to B \to Z) \to \mathit{SF}\ (\mathsf{S}\ A, \mathsf{S}\ B)\ (\mathsf{S}\ Z) \end{aligned}$$

There is no *liftE2*, because there is more than one useful interpretation of such a combinator. Consider: there are two input Event signals, and one output Event signal. At any point in time, if there are event occurrences on both input signals, then it seems that there should be an event occurrence on the output. And if there is no occurrence on either input signal, then it seems there shouldn't be an occurrence on the output signal. But what about when there is an event occurrence on one input signal and not the other?

To address this question, two separate primitives are defined: *merge* and *join*. The behaviour of *merge* is to produce an event occurrence when either input has an occurrence; the behaviour of *join* is to produce an event only when both inputs have an occurrence:

$$\begin{aligned} &\mathit{merge} &&: (A \to Z) \to (B \to Z) \to (A \to B \to Z) \to \mathit{SF}\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z) \\ &\mathit{join} &&: (A \to B \to Z) \to \mathit{SF}\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z) \end{aligned}$$

Finally, *sampleWith* merges an Event signal with a Continuous or Step signal, producing an output event occurrence exactly when there is an occurrence on the input Event signal:

$$\begin{aligned} &\mathit{sampleWithC} &&: (A \to B \to Z) \to \mathit{SF}\ (\mathsf{C}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z) \\ &\mathit{sampleWithS} &&: (A \to B \to Z) \to \mathit{SF}\ (\mathsf{S}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z) \end{aligned}$$

Note that many of these lifting functions are the same other than having differing signal kinds. In an implementation, some form of overloading mechanism might be employed on top of these functions to exploit this.

4.2.4 Primitive Signal Functions

This section introduces the primitive signal functions of N-ary FRP. Their semantic definitions make use of the utility functions from Appendix A.

First, a signal function that emits constant output:

$constantS\ :\ A \rightarrow SF\ as\ (\mathsf{S}\ A)$
$constantS\ a \approx const\ (a, const\ [\,])$

The primitives *never* and *now* generate Event signals:

- *never* generates an Event signal containing no event occurrences;
- *now* generates an Event signal containing exactly one event occurrence at $time_0$.

$never\ :\ SF\ as\ (\mathsf{E}\ A)$
$never \approx const\ (\mathsf{nothing}, const\ [\,])$
$now\ :\ SF\ as\ (\mathsf{E}\ Unit)$
$now \approx const\ (\mathsf{just\ unit}, const\ [\,])$

The primitives *notYet* and *filterE* eliminate selected event occurrences from an Event signal:

- *notYet* eliminates any initial event occurrence;
- *filterE* eliminates any event occurrence for which the given predicate does not hold.

$notYet\ :\ SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)$
$notYet \approx first\ (const\ \mathsf{nothing})$
$filterE\ :\ (A \rightarrow Bool) \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)$

The *hold* and *edge* signal functions mediate between Step and Event signals:

- *hold* emits a Step signal carrying the value of its most recent input event;
- *edge* emits an event whenever the value of the Boolean input Step signal changes from false to true:

```
$hold\ :\ A \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{S}\ A)$
$hold\ a \approx first\ (fromMaybe\ a)$
$edge\ :\ SF\ (\mathsf{S}\ Bool)\ (\mathsf{E}\ Unit)$
$edge \approx \lambda\ (b, cp) \rightarrow (\mathsf{nothing}, edgeAux\ 0\ b \circ cp)$
  where
    $edgeAux\ :\ Time \rightarrow Bool \rightarrow ChangeList\ Bool \rightarrow ChangeList\ Unit$
    $edgeAux\ d\ \_\ \ [\,] = [\,]$
    $edgeAux\ d\ \mathsf{true}\ ((\delta, b) :: \delta bs) = edgeAux\ (d + \delta)\ b\ \delta bs$
    $edgeAux\ d\ \mathsf{false}\ ((\delta, \mathsf{false}) :: \delta bs) = edgeAux\ (d + \delta)\ \mathsf{false}\ \delta bs$
    $edgeAux\ d\ \mathsf{false}\ ((\delta, \mathsf{true}) :: \delta bs) = (d + \delta, \mathsf{unit}) :: edgeAux\ 0\ \mathsf{true}\ \delta bs$
```

Note that *edgeAux* only produces event occurrences in the final case, which is when false is followed by true.

Integrating a Step or Continuous signal always produces a Continuous signal. Any Step signal has a defined integral (see Appendix B.6), but many Continuous signals do not: if the input signal is not integrable, then the semantics of *integralC* applied to that signal is undefined.

$integralS\ :\ SF\ (\mathsf{S}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})$
$integralC\ :\ SF\ (\mathsf{C}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})$
$integralC \approx \lambda\ s\ t_1 \rightarrow \int_0^{t_1} (s\ t)\ dt$

The signal function *when* applies a predicate to a Continuous input signal, producing an event occurrence as output whenever the result changes from false to true. Note that, as with *edge*, this is only at the moment of change: another event will not occur until the predicate has ceased to hold and then become true again.

$when\ :\ (A \to Bool) \to SF\ (\mathsf{C}\ A)\ (\mathsf{E}\ A)$

The *delay* primitives delay a signal by a specified amount of time. Note that in the case of Continuous and Step signals, the signal requires initialising for the delay period:

$delayE\ :\ Time^{+} \to SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)$

$delayS\ :\ Time^{+} \to A \to SF\ (\mathsf{S}\ A)\ (\mathsf{S}\ A)$

$delayC\ :\ Time^{+} \to (Time \to A) \to SF\ (\mathsf{C}\ A)\ (\mathsf{C}\ A)$
$delayC\ d\ f \approx \lambda\ s\ t \to \mathbf{if}\ t < d\ \mathbf{then}\ f\ t\ \mathbf{else}\ s\ (t - d)$

Finally, to allow Step and Continuous signals to be combined, there are two coercion signal functions that convert Step signals to Continuous signals:

$fromS\ \ :\ SF\ (\mathsf{S}\ A)\ (\mathsf{C}\ A)$
$dfromS\ :\ A \to SF\ (\mathsf{S}\ A)\ (\mathsf{C}\ A)$

The difference between the two is that *fromS* defines the value at the moments of change of the resultant Continuous signal to be that of the *new* value of the Step signal (as is also the case for Step signals themselves), whereas *dfromS* defines it to be the *old* value of the Step signal at those moments. One consequence of this is that *dfromS* requires an initial value. (This is similar to the *iPre* signal function found in other FRP variants, see Section 9.1.1.) Signal functions for which changes to their input are not reflected in their output until after that time point are known as *decoupled* signal functions, and are often prefixed with a 'd'. Decoupled signal functions are very important in FRP, as will be discussed in Chapter 7.

4.3 Example N-ary FRP Programs

This section demonstrates N-ary FRP programming by defining some useful signal functions and combinators. Many of these should be familiar from the CFRP and UFRP examples.

4.3.1 Additional Combinators

The routing combinators from UFRP (Section 3.4.6) can be defined as follows:

$toFst\ :\ SF\ as\ cs\ \to SF\ (as, bs)\ cs$
$toFst\ sf\ =\ sfFst \ggg sf$

$toSnd\ :\ SF\ bs\ cs \to SF\ (as, bs)\ cs$
$toSnd\ sf\ =\ sfSnd \ggg sf$

$\_{*}{*}{*}\_\ :\ SF\ as\ cs \to SF\ bs\ ds \to SF\ (as, bs)\ (cs, ds)$
$sf_1 *** sf_2\ =\ toFst\ sf_1\ \&\&\&\ toSnd\ sf_2$

$sfFirst\ :\ SF\ as\ bs \to SF\ (as, cs)\ (bs, cs)$
$sfFirst\ sf\ =\ sf *** identity$

$sfSecond\ :\ SF\ bs\ cs \to SF\ (as, bs)\ (as, cs)$
$sfSecond\ sf\ =\ identity *** sf$

$sfFork\ :\ SF\ as\ (as, as)$
$sfFork\ =\ identity\ \&\&\&\ identity$

$sfSwap\ :\ SF\ (as, bs)\ (bs, as)$
$sfSwap\ =\ sfSnd\ \&\&\&\ sfFst$

$$\begin{aligned}
&\mathit{forkFirst} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathit{as})\\
&\mathit{forkFirst}\ \mathit{sf} = \mathit{sf}\ \&\&\&\ \mathit{identity}
\end{aligned}$$

$$\begin{aligned}
&\mathit{forkSecond} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SF}\ \mathit{as}\ (\mathit{as}, \mathit{bs})\\
&\mathit{forkSecond}\ \mathit{sf} = \mathit{identity}\ \&\&\&\ \mathit{sf}
\end{aligned}$$

Note that the functional level is not used in any of these definitions—the set of routing primitives is sufficient. This is key: as discussed in Section 3.5.2, it is one of the design objectives of N-ary FRP to be able to express all routing at the reactive level.

It can also be useful to re-associate a signal vector:

$$\begin{aligned}
&\mathit{sfAssocL} : \mathit{SF}\ (\mathit{as}, (\mathit{bs}, \mathit{cs}))\ ((\mathit{as}, \mathit{bs}), \mathit{cs})\ \mathsf{cau}\\
&\mathit{sfAssocL} = \mathit{sfSecond}\ \mathit{sfFst}\ \&\&\&\ \mathit{toSnd}\ \mathit{sfSnd}
\end{aligned}$$

$$\begin{aligned}
&\mathit{sfAssocR} : \mathit{SF}\ ((\mathit{as}, \mathit{bs}), \mathit{cs})\ (\mathit{as}, (\mathit{bs}, \mathit{cs}))\ \mathsf{cau}\\
&\mathit{sfAssocR} = \mathit{toFst}\ \mathit{sfFst}\ \&\&\&\ \mathit{sfFirst}\ \mathit{sfSnd}
\end{aligned}$$

Translating the UFRP switching combinators (Section 3.4.6) into N-ary FRP is straightforward:

$$\begin{aligned}
&\mathit{rswitch} : \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A) \to (A \to \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A)) \to \mathit{SF}\ \mathit{as}\ \mathit{bs}\\
&\mathit{rswitch}\ \mathit{sf}\ f = \mathit{switch}\ \mathit{sf}\ (\lambda\ e \to \mathit{rswitch}\ (f\ e \ggg \mathit{sfSecond}\ \mathit{notYet})\ f)
\end{aligned}$$

$$\begin{aligned}
&\mathit{switchWhen} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SF}\ \mathit{bs}\ (\mathsf{E}\ A) \to (A \to \mathit{SF}\ \mathit{as}\ \mathit{bs}) \to \mathit{SF}\ \mathit{as}\ \mathit{bs}\\
&\mathit{switchWhen}\ \mathit{sf}\ \mathit{sfe} = \mathit{switch}\ (\mathit{sf} \ggg \mathit{forkSecond}\ \mathit{sfe})
\end{aligned}$$

$$\begin{aligned}
&\mathit{rswitchWhen} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SF}\ \mathit{bs}\ (\mathsf{E}\ A) \to (A \to \mathit{SF}\ \mathit{as}\ \mathit{bs}) \to \mathit{SF}\ \mathit{as}\ \mathit{bs}\\
&\mathit{rswitchWhen}\ \mathit{sf}\ \mathit{sfe}\ f = \mathit{rswitch}\ (\mathit{sf} \ggg \mathit{forkSecond}\ \mathit{sfe})\ (\lambda\ e \to f\ e \ggg \mathit{forkSecond}\ \mathit{sfe})
\end{aligned}$$

$$\begin{aligned}
&\mathit{replace} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to (A \to \mathit{SF}\ \mathit{as}\ \mathit{bs}) \to \mathit{SF}\ (\mathit{as}, \mathsf{E}\ A)\ \mathit{bs}\\
&\mathit{replace}\ \mathit{sf}\ f = \mathit{rswitch}\ (\mathit{sfFirst}\ \mathit{sf})\ (\lambda\ e \to \mathit{sfFirst}\ (f\ e))
\end{aligned}$$

4.3.2 Library Signal Functions

To demonstrate N-ary FRP programming, this section defines some library signal functions. Constant Continuous signals are often convenient:

$$\begin{aligned}
&\mathit{constantC} : A \to \mathit{SF}\ \mathit{as}\ (\mathsf{C}\ A)\\
&\mathit{constantC}\ a = \mathit{constantS}\ a \ggg \mathit{fromS}
\end{aligned}$$

A decoupled variant of *hold* that emits the value of the most recent event received *before* the current time can be defined as follows:

$$\begin{aligned}
&\mathit{dhold} : A \to \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{C}\ A)\\
&\mathit{dhold}\ a = \mathit{hold}\ a \ggg \mathit{dfromS}\ a
\end{aligned}$$

Initialised versions of integration are defined as usual:

$$\begin{aligned}
&\mathit{iIntegralS} : \mathbb{R} \to \mathit{SF}\ (\mathsf{S}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})\\
&\mathit{iIntegralS}\ x = \mathit{integralS} \ggg \mathit{liftC}\ (+\ x)
\end{aligned}$$

$$\begin{aligned}
&\mathit{iIntegralC} : \mathbb{R} \to \mathit{SF}\ (\mathsf{C}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})\\
&\mathit{iIntegralC}\ x = \mathit{integralC} \ggg \mathit{liftC}\ (+\ x)
\end{aligned}$$

Often when using *sampleWith*, the value of the event is irrelevant:

$$\begin{aligned}
&\mathit{sampleC} : \mathit{SF}\ (\mathsf{C}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ A)\\
&\mathit{sampleC} = \mathit{sampleWithC}\ \mathit{const}
\end{aligned}$$

$$\begin{aligned}
&\mathit{sampleS} : \mathit{SF}\ (\mathsf{S}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ A)\\
&\mathit{sampleS} = \mathit{sampleWithS}\ \mathit{const}
\end{aligned}$$

The local time can be computed by integrating the constant 1:

$$\begin{aligned}
&\mathit{localTime} : \mathit{SF}\ \mathit{as}\ (\mathsf{C}\ \mathit{Time})\\
&\mathit{localTime} = \mathit{constantS}\ 1 \ggg \mathit{integralS}
\end{aligned}$$

Signal functions that emit an event after a specified amount of time, or repeatedly at a fixed interval, are also useful:

$after : Time^{+} \rightarrow SF\ as\ (\mathsf{E}\ Unit)$
$after\ t\ =\ now \ggg delayE\ t$

$repeatedly : Time^{+} \rightarrow SF\ as\ (\mathsf{E}\ Unit)$
$repeatedly\ t\ =\ rswitchWhen\ never\ (after\ t)\ (\lambda\ \_ \rightarrow now)$

There is often a need to assign a new value to an event occurrence, for example:

$tag : A \rightarrow SF\ (\mathsf{E}\ B)\ (\mathsf{E}\ A)$
$tag\ a\ =\ liftE\ (const\ a)$

$nowTag : A \rightarrow SF\ as\ (\mathsf{E}\ A)$
$nowTag\ a\ =\ now \ggg tag\ a$

$afterTag : Time^{+} \rightarrow A \rightarrow SF\ as\ (\mathsf{E}\ A)$
$afterTag\ t\ a\ =\ after\ t \ggg tag\ a$

All but the first occurrence in an Event signal can be suppressed as follows:

$once : SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)$
$once\ =\ switch\ sfFork\ nowTag$

The *fallingBall* signal function from the Bouncing-Ball example is defined:

$fallingBall : Ball \rightarrow SF\ as\ (\mathsf{C}\ Ball)$
$fallingBall\ (h, v)\ =\ constantS\ (-g) \ggg iIntegralS\ v \ggg forkFirst\ (iIntegralC\ h) \ggg liftC2\ (,)$

Note that, unlike in UFRP (see Section 3.4.6), the two output signals have to be explicitly tupled. In UFRP a tuple of signals is identical to a signal of tuples, and thus this explicit tupling is unnecessary.

Except for adapting the type signatures to multi-kinded signals, the other signal functions from the Bouncing-Ball example are the same as in UFRP:

$detectBounce : SF\ (\mathsf{C}\ Ball)\ (\mathsf{E}\ Ball)$
$detectBounce\ =\ when\ detectImpact$

$elasticBall : Ball \rightarrow SF\ as\ (\mathsf{C}\ Ball)$
$elasticBall\ b\ =\ rswitchWhen\ (fallingBall\ b)\ detectBounce\ (fallingBall \circ negateVel)$

$inelasticBall : Ball \rightarrow SF\ as\ (\mathsf{C}\ Ball)$
$inelasticBall\ b\ =\ switchWhen\ (fallingBall\ b)\ detectBounce\ (\lambda\ \_ \rightarrow constantC\ (0,0))$

$resetBall : (Ball \rightarrow SF\ as\ (\mathsf{C}\ Ball)) \rightarrow Ball \rightarrow SF\ (as, \mathsf{E}\ Ball)\ (\mathsf{C}\ Ball)$
$resetBall\ f\ b\ =\ replace\ (f\ b)\ f$

4.4 Totality and Recursion

With the exception of *integralC* and *when*, the semantics of the N-ary FRP primitives is *total*. The semantics of those two signal functions is partial: their arguments are expected to satisfy additional constraints. However, as will be exemplified in Chapter 5, an approximation of these signal functions in a discretely sampled implementation will usually be total without any constraint on the input signal. (Of course, if the host language of an implementation is partial, then partial N-ary FRP programs can always be constructed by lifting partial functions.)

The library N-ary FRP code (Section 4.3) is defined in terms of the N-ary FRP primitives. The definitions are thus all total except for those that use *when* or *integralC*, in which case they are only total if they satisfy the constraints of those primitives. For example, *fallingBall* satisfies the constraint of the input signal to *integralC* being integrable, because that input (the velocity) is a linear function. On the other hand, *iIntegralC* preserves the constraint and is only total if the input signal is integrable. However, in any implementation giving total approximations to *when* and *integralC*, all of the library code would be total.

There is one significant exception to the preceding claim, and that is the *rswitch* combinator. Notice that its definition is *recursive*, and that the recursion is non-terminating (it defines an infinite term). This (potentially ill-defined) use of recursion makes the combinator partial. In particular, *Zeno behaviour* [18, 29] (infinite structural switches in finite time) can be defined. While the use of the *notYet* primitive prevents more than one structural switch at any *individual* time point, it does not prevent infinite switches over a finite interval. This is because the time domain is *dense*: there are infinitely many time points between any two time points. For example, Thomson's Lamp [87, 123] can be encoded as follows:

```
lamp : Time⁺ → Bool → SF as (S Bool)
lamp t b = rswitch (lampAux (t, b)) lampAux
  where
    lampAux : (Time⁺ × Bool) → SF as (S Bool, E (Time⁺ × Bool))
    lampAux (t′, b′) = constantS b′ &&& afterTag t′ (t′/2, not b′)
```

This signal function generates a Step signal that changes ever more rapidly, tending towards infinitely many changes as time tends towards $2t$. This violates the conceptual model of Step signals which requires there to be only a finite number of changes in finite time. Even if a Continuous signal were used instead, there is still a more fundamental problem: the output is undefined from time $2t$ onwards [127].

This problem could be addressed by restricting the *rswitch* combinator in a number of ways, such as requiring a fixed minimum time delta between structural switches. However, the approach taken here is to keep the partial definition, adding the constraint that only uses of *rswitch* that produce a finite number of structural switches in finite time are valid. Note, for example, that the *replace* combinator *does* satisfy this constraint, and thus is total.

One may argue that, having made this decision, the use of *notYet* is unnecessarily restrictive as it does not guarantee totality but does obstruct well-defined uses of *rswitch* that have a finite number (greater than one) of structural switches at some time points. However, there are two pragmatic reasons for including *notYet* in the definition of *rswitch*. First, it is very easy for an FRP programmer to define a combinator such as *replace* and forget to use *notYet* if it has to be done explicitly, inadvertently creating a divergent program. Second, many discretely sampled implementations of FRP will diverge if there are infinitely many structural switches at a time point, but will not do so for a signal function such as *lamp* as the sampling rate will set a minimum rate of switching.

4.5 Conclusions

N-ary FRP is a new FRP variant, inspired heavily by the advantages (and limitations) of UFRP and its Yampa implementation. Like UFRP, it is based around a first-class signal-function abstraction, and does not provide signals (or signal generators) as first-class entities. The main difference between N-ary FRP and UFRP is that N-ary FRP provides *three distinct signal kinds*, and n-ary signal functions which operate over vectors of such *multi-kinded* signals.

The distinction between the signal kinds is enforced by the N-ary FRP type system, thereby ensuring that kind-specific operations cannot be applied to signals of the wrong kind. UFRP does not give this assurance, which can lead to subtle errors in UFRP programs (see Section 5.4 for an example of this). There are existing FRP variants that make a clear distinction between the three signal kinds [37, 63], but, to my knowledge, N-ary FRP is the first FRP variant to do so in a setting with a first-class signal-function abstraction.

A noteworthy advantage of N-ary FRP over most other FRP variants is that N-ary FRP allows all network routing to be expressed at the reactive level. This was an explicit objective, as it allows great flexibility when it comes to implementation and optimisation.

Chapter 5

Embedded Implementations of N-ary FRP

In Section 3.5.1 it was claimed that one of the advantages of an FRP language with signal functions as the primary reactive abstraction is the simplicity and purity with which it can be implemented. However, thus far only conceptual models of FRP have been considered, not any concrete implementations.

This chapter addresses this by embedding an implementation of N-ary FRP in both Agda and Haskell. These primary reason that these two languages are chosen is that they have sufficiently rich type systems to allow N-ary FRP to be defined as an embedded language. The purpose of the Agda embedding is to provide totality and termination guarantees about the implementation, and thus also about N-ary FRP programs. The purpose of the Haskell embedding is to demonstrate that the language and its type system can be embedded in a mainstream functional language that has provided adequate performance for many FRP applications (although this particular implementation is intended only as a proof-of-concept, see Section 8.2 for some discussion as to its efficiency).

The implementation approach is the same in both languages, with the aim of keeping the two embeddings as similar as possible. Neither implementation is optimised, with the emphasis being on clarity rather than efficiency. The complete code for both embeddings is available in the online archive [1].

5.1 Pull-Based Sampling

The implementations in this chapter can be characterised as *pull-based* and *discretely sampled*. This means that the data-flow network (representing the N-ary FRP program) is executed over a discrete sequence of time steps. At each step, a sample of the input signal vector is read in, and then a sample of the output signal vector is emitted. The amount of time that has passed since the previous step is called the *time delta*, and this is implicitly available as an additional input to all signal functions.

Of course, this style of implementation will only approximate the idealised semantics of

N-ary FRP (given in Chapter 4). However, as with other FRP implementations that take this approach, the idea is that the sampling rate will be sufficiently frequent to reduce the error to within some acceptable margin. Discussion of other implementation approaches is postponed until Chapter 8.

Central to the pull-based approach is the notion of a *sample* of a signal vector. A sample is the representation of a signal's value at a specific point in time, such that the (approximation of the) signal comprises its samples at all points in time. The implementations in this chapter use the following representation of samples:

```
Sample : SVDesc → Set
Sample (C A)    = A
Sample (E A)    = Maybe A
Sample (S A)    = A
Sample (as, bs) = Sample as × Sample bs
```

That is, a sample of a Continuous or Step signal is its value, the sample of an Event signal is an optional value, and the sample of a product of signals is a product of samples. The idea behind the sample of an Event signal is that if an event is occurring then the sample is just the value of that event, and if not then the sample is nothing.

This implementation is unoptimised, and thus the same representation is used for samples of Step and Continuous signals. An optimised implementation would likely use a different representation of Step signals to exploit their discrete nature. For example, Step-signal samples could be optional values representing signal *changes*.

In terms of the conceptual model from Section 4.1, the sample of a signal vector at any given time is defined by the following semantic function (not to be confused with the *sampleC* and *sampleS* signal functions):

```
sample : {as : SVDesc} → SigVec as → SampleTime → Sample as
sample {C _} s        t = s t
sample {S _} s        t = val s t
sample {E _} s        t = occ s t
sample {_, _} (s₁, s₂) t = (sample s₁ t, sample s₂ t)
```

The *val* and *occ* utility functions are defined in Appendix B.1.

5.2 Agda Embedding

As discussed, N-ary FRP programs define synchronous data-flow networks. The basic idea of the Agda embedding is to construct a data type to represent such networks, and then define a function over that data type that executes the network for one time step. Running an N-ary FRP program is then achieved by applying this function iteratively.

The embedding is implemented using Agda 2.2.6 with the *--type-in-type* option. This is not a necessary option, but it does simplify the code significantly. Use of this option makes the logic of Agda inconsistent, but that inconsistency is not exploited. An earlier version of this implementation [112] has been encoded without this option to confirm that this is the case.

5.2.1 Network Nodes

The lifting functions (Section 4.2.3) and primitive signal functions (Section 4.2.4) form the nodes of the data-flow network. At each time step the network is provided with an input sample and time delta, and is expected to produce an output sample. Many signal functions are such that their output at any given time point depends upon input from previous time points, which in this case means previous input samples. One way to deal with this would be to have a monolithic global state that stores all past samples, but this would be very inefficient and cause a space leak. Instead, each network node has its own internal state, in which it can store any required information about the past. This state is isolated from the rest of the network: it is only accessible by that node.

A node can therefore be defined as a pair of a state and a transition function that maps a time delta, state, and input sample to an updated state and output sample:

```
data Node (as bs : SVDesc) : Set where
  node : ∀ {Q} → (Δt → Q → Sample as → Q × Sample bs) → Q → Node as bs
```

The identifier Q is used for the (polymorphic) type of the state.

It is then trivial to define a function that executes a node for one time step:

```
stepNode : ∀ {as bs} → Δt → Node as bs → Sample as → Node as bs × Sample bs
stepNode δ (node f q) sa = first (node f) (f δ q sa)
```

5.2.2 Routing

In this style of implementation, there are two possible approaches to network routing: *shallow embedding* and *deep embedding*. Shallow embedding involves encoding the routing at the functional level (i.e. in the host language), whereas deep embedding involves maintaining a representation of the routing at the reactive level.

Shallow Embedding

In this case, a shallow embedding would involve encoding the routing primitives in terms of the existing *Node* data type. For example, *identity* could be encoded as follows:

```
shallowId : ∀ {as} → Node as as
shallowId = node (λ _ _ sa → (unit, sa)) unit
```

As a less trivial example, the &&& routing combinator could be encoded by merging the two component nodes:

```
shallowFan : ∀ {as bs cs} → Node as bs → Node as cs → Node as (bs, cs)
shallowFan {as} {bs} {cs} (node {Q₁} f₁ q₁) (node {Q₂} f₂ q₂) = node fanAux (q₁, q₂)
  where
    fanAux : Δt → Q₁ × Q₂ → Sample as → (Q₁ × Q₂) × Sample (bs, cs)
    fanAux δ (q₁, q₂) sa with f₁ δ q₁ sa | f₂ δ q₂ sa
    ... | (q₁′, sb) | (q₂′, sc) = ((q₁′, q₂′), (sb, sc))
```

The disadvantage of this approach is that once the routing is encoded within the (host language) transition function, the structure of that routing is hidden. As discussed in Section 3.5.2, retaining complete knowledge of the routing at the reactive level is useful for optimisation. Consequently, a deep embedding of the routing primitives is performed instead, as described in the next section. While there is no optimisation in this particular implementation, this approach facilities adding optimisations later.

Listing 5.1 A deep embedding of the primitive combinators

```
data SF : SVDesc → SVDesc → Set where
  prim     : ∀ {as bs}    → Node as bs                      → SF as bs
  arouter  : ∀ {as bs}    → AtomicRouter as bs              → SF as bs
  seq      : ∀ {as bs cs} → SF as bs → SF bs cs             → SF as cs
  fan      : ∀ {as bs cs} → SF as bs → SF as cs             → SF as (bs, cs)
  switcher : ∀ {as bs A}  → SF as (bs, E A) → (A → SF as bs) → SF as bs
  freezer  : ∀ {as bs}    → SF as bs                        → SF as (bs, C (SF as bs))

step : ∀ {as bs} → Δt → SF as bs → Sample as → SF as bs × Sample bs
step δ (prim n) sa = first prim (stepNode δ n sa)
step δ (arouter r) sa = (arouter r, stepARouter r sa)
step δ (seq sf1 sf2) sa with step δ sf1 sa
... | (sf1′, sb) with step δ sf2 sb
... | (sf2′, sc) = (seq sf1′ sf2′, sc)
step δ (fan sf1 sf2) sa with step δ sf1 sa | step δ sf2 sa
... | (sf1′, sb) | (sf2′, sc) = (fan sf1′ sf2′, (sb, sc))
step δ (switcher sf f) sa with step δ sf sa
... | (sf′, (sb, nothing)) = (switcher sf′ f, sb)
... | (_, (_, just e))      = step 0 (f e) sa
step δ (freezer sf) sa with step δ sf sa
... | (sf′, sb) = (freezer sf′, (sb, sf))
```

Deep Embedding

A deep embedding involves constructing an inductive data type to represent the possible network structures, with a constructor for nodes, each routing primitive, and each primitive dynamic combinator. A step function is then defined over this data type to perform one step of network execution (whereas in a shallow embedding the execution of the routing primitives is contained within their definition).

First, to avoid duplication of code later, an auxiliary data type is defined for the three atomic routers: *identity*, *sfFst* and *sfSnd*. Note that to avoid name clashes, the names of the constructors are slightly different to those of the corresponding routing primitives.

```
data AtomicRouter : SVDesc → SVDesc → Set where
  sfId    : ∀ {as}    → AtomicRouter as as
  fstProj : ∀ {as bs} → AtomicRouter (as, bs) as
  sndProj : ∀ {as bs} → AtomicRouter (as, bs) bs
```

A step function for *AtomicRouter* is trivial:

```
stepARouter : ∀ {as bs} → AtomicRouter as bs → Sample as → Sample bs
stepARouter sfId    sa        = sa
stepARouter fstProj (sa1, _)  = sa1
stepARouter sndProj (_, sa2)  = sa2
```

A time delta is not required, and there is no need to return an updated *AtomicRouter* as it contains no state.

Listing 5.1 contains a first attempt at constructing a top-level data type (*SF*), along with a corresponding step function. The most interesting case in the *step* function is for switcher. Notice that if no event occurs, then the output is taken from the subordinate signal function and the switcher is retained. However, if an event does occur, then a residual signal function is generated ($f\ e$), and the *step* function is applied to it immediately with a time delta of 0.

The subordinate signal function, and its output sample, are discarded; only the residual signal function is retained, in place of the entire switching combinator.

There is a problem here: time deltas must be strictly positive, so this is not type correct. Yet the residual signal function needs to be executed immediately, as the output of *switch* is defined (semantically) to be that of the residual signal function at the moment of switching. The alternative is to provide the recursive call to *step* with a time delta other than 0, but that would amount to starting the residual signal function at a different point in time (which would be incorrect). The issue could be avoided by expanding the Δt type to include 0, but a more principled approach is introduced in the next section.

Aside: Coinduction and Infinite Switching

Another issue with the preceding formulation is that it does not allow for infinite recursive switching. For example, a definition of *rswitch* in terms of switcher (as in Section 4.3.1) is rejected by Agda as it forms an infinite term.

Infinite terms can be represented in Agda by using *coinductive data types* [33], so the solution would seem to be to make the type of the switching function coinductive[1]. The definition of *rswitch* in terms of switcher would then be accepted by Agda's termination checker. However, Agda would then (correctly) reject the *step* function as potentially non-terminating. The problem is that, at the moment of switching, the residual signal function starts immediately. Within that residual signal function another structural switch could occur immediately, causing another residual signal function to be generated and immediately start; and so forth. If the switching function is inductive, then there can only be a finite number of such structural switches, as only finite signal-function terms can be expressed. However, if the switching function is coinductive, then the chain of switching could be infinite. For example, a variant of the *rswitch* combinator that did not include a *notYet* in its definition would diverge at the moment of its first structural switch.

The crux of the matter is that the combination of coinductive switching functions and switching combinators that immediately start the residual signal function can lead to non-termination. Consequently, restricting the switching function to be inductive rather than coinductive is beneficial, as it allows an arbitrary yet finite number of structural switches to occur at one time step, while prohibiting an infinite number and thus guaranteeing termination. The *rswitch* combinator is merely a special case of an infinite switching combinator that, while allowing an infinite number of structural switches, restricts there to be at-most one at each time step.

One way to allow *rswitch* would be to introduce a coinductive switching function (as discussed above), and delay the start of all residual signal functions by one time step. However, a simpler solution (and one that avoids introducing delays) is to make *rswitch* the primitive switching combinator instead of *switch*. Once *rswitch* is a primitive, the restriction that there must be at-most one structural switch per time step can easily be incorporated into the step function. This is accepted by Agda's termination checker, and it is then easy to define *switch* in terms of *rswitch*:

[1]A reader familiar with Agda can note that the type of the *switch* combinator would then be: $SF\ as\ (bs, \mathsf{E}\ e) \rightarrow (e \rightarrow \infty\ (SF\ as\ bs)) \rightarrow SF\ as\ bs$

Listing 5.2 Uninitialised and initialised signal functions

```
data SF  : SVDesc → SVDesc → Set where
  prim      : ∀ {as bs}    → (Sample as → Node as bs × Sample bs)  → SF as bs
  arouter   : ∀ {as bs}    → AtomicRouter as bs                    → SF as bs
  seq       : ∀ {as bs cs} → SF as bs → SF bs cs                   → SF as cs
  fan       : ∀ {as bs cs} → SF as bs → SF as cs                   → SF as (bs, cs)
  rswitcher : ∀ {as bs A}  → SF as (bs, E A) → (A → SF as (bs, E A)) → SF as bs
  freezer   : ∀ {as bs}    → SF as bs                              → SF as (bs, C (SF as bs))

data SF′ : SVDesc → SVDesc → Set where
  prim      : ∀ {as bs}    → Node as bs                             → SF′ as bs
  arouter   : ∀ {as bs}    → AtomicRouter as bs                     → SF′ as bs
  seq       : ∀ {as bs cs} → SF′ as bs → SF′ bs cs                  → SF′ as cs
  fan       : ∀ {as bs cs} → SF′ as bs → SF′ as cs                  → SF′ as (bs, cs)
  rswitcher : ∀ {as bs A}  → SF′ as (bs, E A) → (A → SF as (bs, E A)) → SF′ as bs
  freezer   : ∀ {as bs}    → SF′ as bs                              → SF′ as (bs, C (SF as bs))
```

```
switch : ∀ {as bs A} → SF as (bs, E A) → (A → SF as bs) → SF as bs
switch sf f = rswitch sf (λ e → f e &&& never)
```

Thus, for the refinement of the implementation in the next section, an rswitcher constructor is used instead of switcher.

5.2.3 A Distinct Initialisation Step

The first step of execution of a signal function is a special case, as there are no previous input samples at that point. Also, as discussed in the previous section, the first step of execution can be at (local) $time_0$, in which case there is no time delta either.

This is addressed by making the design decision that the first step of execution will *always* be at $time_0$, and that this step will be distinct from all other steps. This step is called the *initialisation step*. The *SF* type and *step* function in Listing 5.1 are insufficient to express this (as they treat all steps the same), and so they need to be refined.

First, the *SF* data type in Listing 5.1 is replaced with the two data types in Listing 5.2. The *SF* data type now represents *uninitialised* signal functions (those that have not yet undergone an initialisation step), and the new SF' data type represents *initialised* signal functions (those that have undergone an initialisation step and are now considered to be "running"). The key difference between *SF* and SF' is the prim constructor. In *SF* it contains a function that, given an input sample, produces a *Node* and an output sample. Thus it can produce the initial output with no dependence on a time delta or an internal state (recall that the internal state represents past inputs). Also, note that in SF', the new subordinate signal function generated by the switching function is an uninitialised signal function. Making this explicit in the type is an additional advantage of this representation.

The *step* function in Listing 5.1 is likewise replaced by the two functions in Listing 5.3. The initialisation step is represented by $step_0$, which converts an *SF* to an SF' and does not take a time delta as an argument. All other steps are represented by $step'$, which is almost identical to the original *step* function, except in two places. The first is the recursive call made after a structural switch occurs, which is made to $step_0$ rather than $step'$ thereby avoiding the need for a time delta. The second is in the case of freezer, which uses the auxiliary function *freezeSF* to

Listing 5.3 Step functions with an initialisation step

```
step0 : ∀ {as bs} → SF as bs → Sample as → SF' as bs × Sample bs
step0 (prim f) sa = first prim (f sa)
step0 (arouter r) sa = (arouter r, stepARouter r sa)
step0 (seq sf1 sf2) sa with step0 sf1 sa
... | (sf'1, sb) with step0 sf2 sb
... | (sf'2, sc) = (seq sf'1 sf'2, sc)
step0 (fan sf1 sf2) sa with step0 sf1 sa | step0 sf2 sa
... | (sf'1, sb) | (sf'2, sc) = (fan sf'1 sf'2, (sb, sc))
step0 (rswitcher sf f) sa with step0 sf sa
... | (sf', (sb, nothing)) = (rswitcher sf' f, sb)
... | (_, (_, just e)) with step0 (f e) sa
... | (sf', (sb, _)) = (rswitcher sf' f, sb)
step0 (freezer sf) sa with step0 sf sa
... | (sf', sb) = (freezer sf', (sb, sf))

step' : ∀ {as bs} → Δt → SF' as bs → Sample as → SF' as bs × Sample bs
step' δ (prim n) sa = first prim (stepNode δ n sa)
step' δ (arouter r) sa = (arouter r, stepARouter r sa)
step' δ (seq sf1 sf2) sa with step' δ sf1 sa
... | (sf'1, sb) with step' δ sf2 sb
... | (sf'2, sc) = (seq sf'1 sf'2, sc)
step' δ (fan sf1 sf2) sa with step' δ sf1 sa | step' δ sf2 sa
... | (sf'1, sb) | (sf'2, sc) = (fan sf'1 sf'2, (sb, sc))
step' δ (rswitcher sf f) sa with step' δ sf sa
... | (sf', (sb, nothing)) = (rswitcher sf' f, sb)
... | (_, (_, just e)) with step0 (f e) sa
... | (sf', (sb, _)) = (rswitcher sf' f, sb)
step' δ (freezer sf) sa with step' δ sf sa
... | (sf', sb) = (freezer sf', (sb, freezeSF δ sf))
  where
    freezeSF : ∀ {as bs} → Δt → SF' as bs → SF as bs
    freezeSF δ (prim n)         = prim (stepNode δ n)
    freezeSF δ (arouter r)      = arouter r
    freezeSF δ (seq sf1 sf2)    = seq (freezeSF δ sf1) (freezeSF δ sf2)
    freezeSF δ (fan sf1 sf2)    = fan (freezeSF δ sf1) (freezeSF δ sf2)
    freezeSF δ (rswitcher sf f) = rswitcher (freezeSF δ sf) f
    freezeSF δ (freezer sf)     = freezer (freezeSF δ sf)
```

convert the SF' back into an SF, by providing it with the time delta. This is another advantage of the two distinct SF types: *freezeSF* is forced to correctly take into account the time delta (in Listing 5.1 the frozen signal function incorrectly ignored the time delta).

5.2.4 Primitives

The necessary infrastructure to represent and execute a network of signal functions has now been established. This section completes the Agda embedding of N-ary FRP by defining the N-ary FRP primitives in terms of that infrastructure.

Routing Primitives and Dynamic Combinators

As the implementation uses a deep embedding of the routing primitives and dynamic combinators, their definitions are trivial:

$identity : \forall \{as\} \to SF\ as\ as$
$identity = \mathsf{arouter\ sfId}$

$sfFst : \forall \{as\ bs\} \to SF\ (as, bs)\ as$
$sfFst = \mathsf{arouter\ fstProj}$

$sfSnd : \forall \{as\ bs\} \to SF\ (as, bs)\ bs$
$sfSnd = \mathsf{arouter\ sndProj}$

$\_\ggg\_ : \forall \{as\ bs\ cs\} \to SF\ as\ bs \to SF\ bs\ cs \to SF\ as\ cs$
$\_\ggg\_ = \mathsf{seq}$

$\_\&\&\&\_ : \forall \{as\ bs\ cs\} \to SF\ as\ bs \to SF\ as\ cs \to SF\ as\ (bs, cs)$
$\_\&\&\&\_ = \mathsf{fan}$

$rswitch : \forall \{as\ bs\ A\} \to SF\ as\ (bs, \mathsf{E}\ A) \to (A \to SF\ as\ (bs, \mathsf{E}\ A)) \to SF\ as\ bs$
$rswitch = \mathsf{rswitcher}$

$freeze : \forall \{as\ bs\} \to SF\ as\ bs \to SF\ as\ (bs, \mathsf{C}\ (SF\ as\ bs))$
$freeze = \mathsf{freezer}$

Utilities for Defining Primitives

Before defining the rest of the primitives, some utility functions are first introduced. These are not part of N-ary FRP (and thus will be hidden from the N-ary FRP programmer); they are merely for internal use in defining the primitives.

First, to increase the readability of the code, some synonyms are defined for the *Maybe* constructors that will be used when dealing with event samples:

$noEvent : \forall \{A\} \to Sample\ (\mathsf{E}\ A)$
$noEvent = \mathsf{nothing}$

$event : \forall \{A\} \to A \to Sample\ (\mathsf{E}\ A)$
$event = \mathsf{just}$

The lifting functions (Section 4.2.3) and primitive signal functions (Section 4.2.4) are defined using the prim constructor. However, many of them have similarities that lead to recurring patterns in their definitions. To exploit this, some construction functions are introduced that abstract out these common patterns. Note that these common patterns will appear again in Section 8.3.3, where they will form the basis for optimisations.

First, a general construction function is defined as follows:

$mkSF : \forall \{as\ bs\ Q\} \to (\Delta t \to Q \to Sample\ as \to Q \times Sample\ bs) \to$
$\qquad (Sample\ as \to Q \times Sample\ bs) \to SF\ as\ bs$
$mkSF\ f\ g = \mathsf{prim}\ (first\ (\mathsf{node}\ f) \circ g)$

As discussed, signal generators can be represented in N-ary FRP as signal functions that ignore their input. Such signal functions are called *sources*. Constructing a source requires an initial output sample, an initial state, and a state update function that produces an output sample after each subsequent time step:

$mkSFsource : \forall \{as\ bs\ Q\} \to (\Delta t \to Q \to Q \times Sample\ bs) \to Q \to Sample\ bs \to SF\ as\ bs$
$mkSFsource\ f\ q\ sb = mkSF\ (\lambda\ \delta\ q'\ \_ \to f\ \delta\ q')\ (const\ (q, sb))$

Some signal functions do not depend upon time, but merely the order in which samples are received. These are called *timeless* signal functions, and they can be defined by an initial state and a transition function that does not take a time delta as an argument:

$mkSFtimeless : \forall \{as\ bs\ Q\} \to (Q \to Sample\ as \to Q \times Sample\ bs) \to Q \to SF\ as\ bs$
$mkSFtimeless\ f\ q = mkSF\ (const\ f)\ (f\ q)$

Some signal functions do not have an internal state, and do not depend on time. These are called *stateless* signal functions, and are a subset of the timeless signal functions. They can be constructed from a function mapping an input sample to an output sample. Note that the present implementation requires a state even when not used, so a unit state is used:

$mkSFstateless\ :\ \forall\ \{as\ bs\} \rightarrow (Sample\ as \rightarrow Sample\ bs) \rightarrow SF\ as\ bs$
$mkSFstateless\ f\ =\ mkSFtimeless\ (\lambda\ \_\ sa \rightarrow (\mathsf{unit}, f\ sa))\ \mathsf{unit}$

Finally, a subset of the stateless signal functions are the *changeless* (constant) signal functions, those that produce the same output sample at every time step:

$mkSFchangeless\ :\ \forall\ \{as\ bs\} \rightarrow Sample\ bs \rightarrow SF\ as\ bs$
$mkSFchangeless\ sb\ =\ mkSFstateless\ (const\ sb)$

Primitive Signal Functions

Defining the primitive signal functions is now (mostly) straightforward. First *constantS* and *never* are easily defined in terms of *mkSFchangeless*:

$constantS\ :\ \forall\ \{as\ A\} \rightarrow A \rightarrow SF\ as\ (\mathsf{S}\ A)$
$constantS\ a\ =\ mkSFchangeless\ a$

$never\ :\ \forall\ \{as\ A\} \rightarrow SF\ as\ (\mathsf{E}\ A)$
$never\ =\ mkSFchangeless\ noEvent$

The *now* signal function is slightly more complicated as it produces an initial event. However, thereafter it will always produce *noEvent*, so a unit state and a constant transition function are used:

$now\ :\ \forall\ \{as\} \rightarrow SF\ as\ (\mathsf{E}\ Unit)$
$now\ =\ mkSFsource\ (\lambda\ \_\ \_ \rightarrow (\mathsf{unit}, noEvent))\ \mathsf{unit}\ (event\ \mathsf{unit})$

The *notYet* signal function produces *noEvent* initially, and is an identity function thereafter:

$notYet\ :\ \forall\ \{A\} \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)$
$notYet\ =\ mkSF\ (\lambda\ \_ \rightarrow curry\ id)\ (const\ (\mathsf{unit}, noEvent))$

The *filterE* signal function is a stateless signal function that filters the event samples:

$filterE\ :\ \forall\ \{A\} \rightarrow (A \rightarrow Bool) \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)$
$filterE\ p\ =\ mkSFstateless\ (maybeFilter\ p)$

The *hold* signal function stores the most recent input event occurrence in its internal state. The state is emitted as the output (after updating it, if necessary) at every time step:

$hold\ :\ \forall\ \{A\} \rightarrow A \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{S}\ A)$
$hold\ =\ mkSFtimeless\ (\lambda\ q \rightarrow fork \circ fromMaybe\ q)$

The signal functions *edge* and *when* are very similar. Their internal state is a Boolean that records whether the previous input (or predicate applied to the previous input) was true. That state is compared with the current input to see if an event should be emitted:

$edge\ :\ SF\ (\mathsf{S}\ Bool)\ (\mathsf{E}\ Unit)$
$edge\ =\ mkSFtimeless\ (\lambda\ q\ i \rightarrow (i, (\mathbf{if}\ i\ \&\&\ not\ q\ \mathbf{then}\ event\ \mathsf{unit}\ \mathbf{else}\ noEvent)))\ \mathsf{true}$
$when\ :\ \forall\ \{A\} \rightarrow (A \rightarrow Bool) \rightarrow SF\ (\mathsf{C}\ A)\ (\mathsf{E}\ A)$
$when\ p\ =\ mkSFtimeless\ (\lambda\ q\ i \rightarrow (p\ i, (\mathbf{if}\ p\ i\ \&\&\ not\ q\ \mathbf{then}\ event\ i\ \mathbf{else}\ noEvent)))\ \mathsf{true}$

Integration is defined using the rectangle rule on Step signals, and using the trapezium rule on Continuous signals. The internal state required for this is a pair of the current total and the most recent input sample.

Listing 5.4 Embedding the lifting functions

```
liftC : ∀ {A B} → (A → B) → SF (C A) (C B)
liftC = mkSFstateless

liftS : ∀ {A B} → (A → B) → SF (S A) (S B)
liftS = mkSFstateless

liftE : ∀ {A B} → (A → B) → SF (E A) (E B)
liftE = mkSFstateless ∘ maybeMap

liftC2 : ∀ {A B Z} → (A → B → Z) → SF (C A, C B) (C Z)
liftC2 = mkSFstateless ∘ uncurry

liftS2 : ∀ {A B Z} → (A → B → Z) → SF (S A, S B) (S Z)
liftS2 = mkSFstateless ∘ uncurry

merge : ∀ {A B Z} → (A → Z) → (B → Z) → (A → B → Z) → SF (E A, E B) (E Z)
merge f_a f_b f_ab = mkSFstateless (uncurry (maybeMerge f_a f_b f_ab))

join : ∀ {A B Z} → (A → B → Z) → SF (E A, E B) (E Z)
join = mkSFstateless ∘ uncurry ∘ maybeMap2

sampleWithC : ∀ {A B Z} → (A → B → Z) → SF (C A, E B) (E Z)
sampleWithC f = mkSFstateless (uncurry (maybeMap ∘ f))

sampleWithS : ∀ {A B Z} → (A → B → Z) → SF (S A, E B) (E Z)
sampleWithS f = mkSFstateless (uncurry (maybeMap ∘ f))
```

```
IntegralState = ℝ × ℝ

integrateRectangle : Δt → IntegralState → ℝ → IntegralState × ℝ
integrateRectangle δ (tot, x₁) x₂ = let tot′ = tot + (δ * x₁)
                                    in ((tot′, x₂), tot′)

integrateTrapezium : Δt → IntegralState → ℝ → IntegralState × ℝ
integrateTrapezium δ (tot, x₁) x₂ = let tot′ = tot + (δ * (x₁ + x₂) / 2)
                                    in ((tot′, x₂), tot′)

integralS : SF (S ℝ) (C ℝ)
integralS = mkSF integrateRectangle (λ x₀ → ((0, x₀), 0))

integralC : SF (C ℝ) (C ℝ)
integralC = mkSF integrateTrapezium (λ x₀ → ((0, x₀), 0))
```

As *Step* and *Continuous* signals have the same representation in this implementation, the *fromS* signal function is just an identity function:

```
fromS : ∀ {A} → SF (S A) (C A)
fromS = mkSFstateless id
```

Whereas *dfromS* is implemented as a one-time-step delay:

```
dfromS : ∀ {A} → A → SF (S A) (C A)
dfromS = mkSFtimeless (flip _,_)
```

Finally, the *delay* signal functions are omitted as they are substantially more involved than the other primitives. Their definitions can be found in Appendix C.1. However, an informal overview of their implementation is given in Section 5.4, as it provides a good example of the benefits of multi-kinded signals.

Lifting Functions

The many lifting functions of N-ary FRP can be defined as stateless signal functions that are just pointwise mappings over the input samples. In the case of products of signals this involves uncurrying the lifted function, and in the case of events there is additional mapping over *Maybe* types. The definitions are very similar, and can be found in Listing 5.4.

5.3 Haskell Embedding

Much of the code for the Haskell embedding described in this section corresponds closely to the Agda code in the preceding section. Consequently, only the noteworthy differences are discussed and the majority of the code is relegated to Appendix C.2.

5.3.1 Language Extensions

The version of Haskell used for this implementation is Haskell 2010 [86], with several language extensions provided by the Glasgow Haskell Compiler (GHC) [121].

The extensions used, along with their corresponding flags, are:

- Empty Data Declarations (*EmptyDataDecls*)
- Generalised Algebraic Data Types (*GADTs*) [64]
- Kind Signatures (*KindSignatures*)
- Scoped Type Variables (*ScopedTypeVariables*)
- Type Families (*TypeFamilies*) [66, 109]

These extensions provide most of the features of the Agda type system that were required for the Agda embedding. The key ideas are to use GADTs instead of Agda data types, empty data types instead of type-level values, and type families instead of type-level functions [66, 116]. This is slightly more awkward, and less type safe, than the Agda embedding, but it allows the N-ary FRP type system to be encoded.

5.3.2 Signals and Samples

The first task is encoding signal vector descriptors. Unlike Agda, data cannot be used at the type level in Haskell. However, type-level values can be simulated by empty data types:

```
data C a :: *
data E a :: *
data S a :: *
```

Unfortunately, GHC does not yet support *algebraic data kinds* [114], and so all such type-level values have to be assigned the kind $*$. This is less type safe than Agda, but does have the convenient side effect that the Haskell product type can be used for products of signals, rather than having to define a separate type constructor. For example, the type-level function *Sample* can be defined (using type families) as follows:

```
type family   Sample as       :: *
type instance Sample (C a)    = a
type instance Sample (E a)    = Maybe a
type instance Sample (S a)    = a
type instance Sample (as, bs) = (Sample as, Sample bs)
```

5.3.3 Time and Recursion

To make the implementation executable a computable representation of time is needed. In this case, double-precision floating-point numbers are used:

```
type Time = Double
type Dt   = Time
```

The remainder of the Haskell embedding is sufficiently similar to the Agda code that it is relegated to Appendix C.2. However, there is one noteworthy distinction. Recall from Section 5.2.2 that the Agda embedding made *rswitch* a primitive rather than *switch*, because the definition of *rswitch* in terms of *switch* is not accepted by Agda's termination checker. This is not necessary in Haskell, as non-termination and infinite terms are permitted. Therefore, in the Haskell embedding, *switch* is taken as a primitive (which is preferable as *switch* is simpler and easier to optimise).

5.3.4 Interaction with the Outside World

The *step* functions are not a full execution mechanism, but constitute the core of one. Running an N-ary FRP program involves applying the step functions iteratively, reading some input and emitting some output at each iteration.

Designing interfaces for FRP variants is somewhat orthogonal to the topics of this thesis. However, to give one example, consider the following function that executes a network at as rapid a sampling rate as the system resources available to it will allow:

```
runSF :: forall as bs.SF as bs → IO (Sample as) → (Sample bs → IO ()) → IO Time → IO Bool → IO ()
runSF sf ins outs time done = do sa ← ins
                                 let (sf′, sb) = step0 sf sa
                                 outs sb
                                 runSF′ 0 sf′
  where
    runSF′ :: Time → SF′ as bs → IO ()
    runSF′ t0 sf′ = do sa ← ins
                       t1 ← time
                       let (sf″, sb) = step′ (t1 − t0) sf′ sa
                       outs sb
                       d ← done
                       unless d (runSF′ t1 sf″)
```

The basic idea is that the signal function is executed iteratively, until some termination condition is met. At each iteration, an input sample is read in, and an output sample is emitted. The arguments to the *runSF* function are: the signal function to execute (*sf*), an *IO* action to read an input sample (*ins*), an *IO* action to emit an output sample (*outs*), an *IO* action to read the current time (*time*), and an *IO* action to determine if execution should terminate (*done*).

Finally, note that an N-ary FRP library would be expected to provide additional infrastructure on top of such a function.

5.4 Delaying Signals

Recall the family of *delay* signal functions:

$$\begin{array}{lll} \mathit{delayC} & : \mathit{Time}^{+} \to (\mathit{Time} \to A) & \to \mathit{SF}\ (\mathsf{C}\ A)\ (\mathsf{C}\ A) \\ \mathit{delayS} & : \mathit{Time}^{+} \to A & \to \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{S}\ A) \\ \mathit{delayE} & : \mathit{Time}^{+} & \to \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{E}\ A) \end{array}$$

Intuitively, these signal functions delay a signal by a specified amount of time. The Agda embedding of these signal functions can be found in Appendix C.1. This section provides a rough overview of those implementations, and discusses the benefits multi-kinded signals bring to ensuring that those implementations respect the abstractions of the different signal kinds.

The basic idea is to store the current time and a queue of samples in the internal state of the signal function. At every time step, the input sample is added to the queue along with a time stamp recording when it should be dequeued (the current time plus the delay period). While the current time is less than the delay period, the output is determined depending upon the signal kind:

- a Continuous signal applies the initialisation function to the current time;
- a Step signal uses the initialisation value; and
- an Event signal has no event occurrence.

However, determining the output sample after the delay period is a little more complicated.

First consider the simple case where the time delta is constant. Once the current time reaches the delay period, the first sample is dequeued and used as the output sample. Thereafter, at each time step the next sample is dequeued and emitted.

However, if the time delta varies then at any given time there may be zero, one, or more samples ready to be dequeued. In each case a single output sample has to be emitted. For Step and Continuous signals, if no sample is yet available for dequeueing this can be dealt with by repeating the previous output sample, and continuing to do so until a sample is ready to be dequeued. This is called *oversampling*[2]. On the other hand, if more than one sample is ready to be dequeued, then the most recent sample can be used and the rest discarded. This is called *undersampling*.

Event signals are more problematic. One could imagine using the same technique of oversampling and undersampling, but this would not do what the N-ary FRP programmer expects. Conceptually, events are isolated instantaneous occurrences; delaying an Event signal should not change the number of events, merely the time points at which they occur. In particular, the sampling rate (which is hidden from the N-ary FRP programmer) should not cause events to be gained and lost in this fashion[3]. Thus duplicating or eliminating samples containing event occurrences is unacceptable, though samples without an event occurrence could be duplicated or eliminated freely.

[2]The terminology comes from the synchronous data-flow languages, and should not be confused with oversampling in the field of signal processing.

[3]Of course, the sampling rate will always introduce imprecision relative to the ideal semantics. However, this is not just a margin of inaccuracy. Here, a small change to the sampling rate or delay period could easily cause all event occurrences to be lost from an Event signal.

This is dealt with by only storing event *occurrences* in the queue. The output sample is then determined by the number of event occurrences that are ready to be dequeued. If there are no events ready to be dequeued, the absence of an event is emitted. If there is exactly one event ready to be dequeued, then that event is emitted. However, if there are several events ready to be dequeued, then things are trickier. For this situation to arise, events need to be occurring faster than the sampling rate, which is usually a sign that the sampling rate is insufficient for the application. Still, the implementation should attempt to deal with it. There are really two options: either emit the most recent event and discard the remainder (a form of undersampling, though not quite as detrimental as that previously discussed), or emit the head of the queue and retain the rest for future time steps. The latter option ensures that event occurrences are never lost, but it has the potential to build up an increasing backlog of event occurrences. It relies on the sampling rate *on average* exceeding the event-occurrence rate, thereby allowing any backlog of events to be cleared. The embeddings described in this chapter use this latter option.

The preceding discussion should have demonstrated that it can be important to give different treatment to the different signal kinds. This is one of the motivations for a multi-kinded conceptual model. By way of comparison, consider the UFRP-based Yampa implementation, where events are embedded in conceptually continuous-time signals. The *delay* signal function is designed for continuous-time signals, and thus performs oversampling and undersampling. However, being polymorphic in its signal type, it can also be applied to event signals. This leads to the elimination and duplication of event occurrences. Yampa does also provide a specialised signal function for delaying event signals, which behaves as *delayE*. However, this relies on the programmer remembering to use this instead of the standard *delay* when dealing with event signals. This can lead to subtle errors in Yampa programs.

5.5 Conclusions

This chapter defined embedded implementations of the N-ary FRP language within both Agda and Haskell. As Agda checks the totality and termination of all functions, the (Agda) implementation is guaranteed to be *productive* [118]. That is, execution of a signal function by repeated application of the step function will produce each output sample within finite time, and without run-time errors. The same guarantee is not provided by the Haskell implementation, but the close correspondence between the two can at least provide a degree of confidence in the Haskell embedding.

The design of these implementations is quite similar to that of Yampa [92], which is unsurprising given they both take signal functions to be the primary reactive abstraction. The main difference is that Yampa uses a shallow embedding for the routing primitives and dynamic combinators (rather than the deep embedding employed here). The latest version of Yampa also contains a significant amount of optimisations, which complicate the implementation somewhat [90]. However, in essence, Yampa is based around a core data type containing a transition function:

```
data SF a b = SF (Dt → a → (SF a b, b))
```

This continuation-style transition function is expressive enough to encapsulate any internal state, so the type of the state does not need to be made explicit. N-ary FRP could be implemented in this same style in both Haskell and Agda. In the Agda embedding this would require explicit use of coinduction (as it is a coinductive data type), and *rswitch* would again have to be a primitive to be accepted by Agda's termination checker (for essentially the same reasons as discussed in Section 5.2.2).

Finally, this chapter discussed the *delay* signal functions as an example of how multi-kinded signals allow an implementation to better respect the conceptual model. Single-kinded models that embed event signals in continuous-time signals cannot protect the abstractions of event signals to the same extent.

Chapter 6

Temporal Logic

Central to FRP is the notion of time-varying values called signals. Reasoning about signals can be facilitated by using concepts from *temporal logic* [124], which is the logic of time-varying properties. This chapter gives a brief introduction to temporal logic, introduces a selection of temporal operators, and then uses them to define a number of properties of signal functions pertaining to the N-ary FRP setting.

6.1 Introduction

Temporal logics are modal logics in which propositions are quantified over time. For example, assuming some proposition Φ, typical temporal logic statements might be "eventually it will be the case that Φ", or "Φ has always been the case".

A temporal logic consists of an underlying time domain of quantification, a set of time-dependent propositions, and a set of temporal operators that extend propositions over time. Further propositions and operators can then be defined in terms of these primitives, allowing time-dependent properties to be expressed intuitively and concisely.

Temporal logic is a well-established means of specifying and reasoning about reactive systems [65, 81, 83, 85]. The approach taken in this thesis is not to work within any particular temporal logic, but to instead take several well-known temporal operators and define them directly as logical combinators. Some properties will be expressed entirely by these operators (i.e. purely in temporal logic), while others will only be partially defined by them.

Temporal logics vary depending on the nature of the underlying time domain. Time can be either continuous or discrete, of finite or infinite duration, and either branching, linear or cyclic in shape. A linear or branching time domain may (or may not) have a first point or an end point. The (conceptual) time domain of FRP is that of the non-negative real numbers. This is a continuous, linear and infinite domain with a first point but no end point.

6.2 Temporal Operators

Temporal logic is concerned with time-varying properties; that is, properties that hold at some points in time but not others. These *temporal predicates* are formulated as follows:

$TPred \;=\; Time \to Set$

Temporal operators map temporal predicates to temporal predicates. Some properties can be expressed *entirely* by such operators, which avoids having to explicitly mention any time *values* in their definitions.

The required temporal operators can be divided into two groups: *lifted logical operators* and *Priorean operators*.

6.2.1 Lifted Logical Operators

The standard logical operators can be lifted to the temporal logic level in a pointwise fashion. For example, a temporal disjunction ($\varphi \vee \psi$) holds at any point in time at which either of its sub-formulae hold:

$$\begin{aligned}&\_\!\vee\!\_ \;:\; TPred \to TPred \to TPred\\&(\varphi \vee \psi)\; t \;=\; \varphi\; t \uplus \psi\; t\end{aligned}$$

Falsehood, truth, conjunction and implication are lifted in a similar manner:

$$\begin{aligned}&\bot \;:\; TPred\\&\bot\; t \;=\; False\\&\top \;:\; TPred\\&\top\; t \;=\; True\\&\_\!\wedge\!\_ \;:\; TPred \to TPred \to TPred\\&(\varphi \wedge \psi)\; t \;=\; \varphi\; t \times \psi\; t\\&\_\!\Rightarrow\!\_ \;:\; TPred \to TPred \to TPred\\&(\varphi \Rightarrow \psi)\; t \;=\; \varphi\; t \to \psi\; t\end{aligned}$$

New temporal operators can be constructed from these operators. For example, time-varying negation can be defined as:

$$\begin{aligned}&\neg\_ \;:\; TPred \to TPred\\&\neg\,\varphi \;=\; \varphi \Rightarrow \bot\end{aligned}$$

6.2.2 Priorean Operators

The operators discussed so far only allow temporal predicates to be combined pointwise; that is, the truth of the composite formula at any point in time depends only upon the truth of its component formulae at that time point. This section introduces operators that refer to the past and future. Specifically, the four Priorean temporal operators are considered (originally conceived by Prior [107]). These are unary operators, and are called *Past*, *Future*, *History* and *Global*. For consistency with temporal-logic literature, their denotations will be typeset in bold.

Note that each Priorean operator has a *mirror-image* operator that is equivalent other than that the ordering of time is reversed. Thus, an operator can be converted to its mirror image by replacing all occurrences of less-than in its definition with greater-than.

The *Past* operator is denoted **P**, and should be read as "at some point in the past". That is, $\mathbf{P}\,\varphi$ holds if φ held previously. This is encoded as a dependent product: the point in time at which φ held, the proof that it is an earlier time point, and the proof of φ at that time point:

$$\begin{aligned}&\mathbf{P} \;:\; TPred \to TPred\\&\mathbf{P}\,\varphi\; t \;=\; \Sigma\; Time\; (\lambda\, t' \to (t' < t) \times \varphi\; t')\end{aligned}$$

The *Future* operator is denoted **F**, and should be read as "at some point in the future". It is defined as the mirror image of *Past*:

$$\begin{array}{l} \mathbf{F} \;:\; \mathit{TPred} \to \mathit{TPred} \\ \mathbf{F}\,\varphi\;t \;=\; \Sigma\;\mathit{Time}\;(\lambda\;t' \to (t' > t) \times \varphi\;t') \end{array}$$

The *Global* and *History* operators are denoted **G** and **H**, and are mirror images of each other. They should be read as "at all points in the future" and "at all points in the past", respectively. They are encoded as dependent functions:

$$\begin{array}{l} \mathbf{G} \;:\; \mathit{TPred} \to \mathit{TPred} \\ \mathbf{G}\,\varphi\;t \;=\; (t' \;:\; \mathit{Time}) \to t' > t \to \varphi\;t' \end{array}$$

$$\begin{array}{l} \mathbf{H} \;:\; \mathit{TPred} \to \mathit{TPred} \\ \mathbf{H}\,\varphi\;t \;=\; (t' \;:\; \mathit{Time}) \to t' < t \to \varphi\;t' \end{array}$$

Note that these temporal operators are *strict*: they exclude the current time from their domain of quantification. In many temporal logics these operators are instead defined non-strictly, and in this thesis both are needed. One approach would be to define reflexive (non-strict) variants of these operators with similar semantic definitions to their strict counterparts, except using at-most and at-least instead of less-than and greater-than. However, it is preferable to define these reflexive variants at the temporal logic level, in terms of the existing operators:

$$\begin{array}{l} \mathbf{F^r} \;:\; \mathit{TPred} \to \mathit{TPred} \\ \mathbf{F^r}\,\varphi \;=\; \varphi \vee \mathbf{F}\,\varphi \end{array}$$

$$\begin{array}{l} \mathbf{P^r} \;:\; \mathit{TPred} \to \mathit{TPred} \\ \mathbf{P^r}\,\varphi \;=\; \varphi \vee \mathbf{P}\,\varphi \end{array}$$

$$\begin{array}{l} \mathbf{G^r} \;:\; \mathit{TPred} \to \mathit{TPred} \\ \mathbf{G^r}\,\varphi \;=\; \varphi \wedge \mathbf{G}\,\varphi \end{array}$$

$$\begin{array}{l} \mathbf{H^r} \;:\; \mathit{TPred} \to \mathit{TPred} \\ \mathbf{H^r}\,\varphi \;=\; \varphi \wedge \mathbf{H}\,\varphi \end{array}$$

6.3 Introducing and Eliminating Temporal Predicates

As a means of introducing temporal predicates, a *time-varying equality* is defined by lifting propositional equality pointwise over time-varying values:

$$\begin{array}{l} \_ \doteq \_ \;:\; (\mathit{Time} \to A) \to (\mathit{Time} \to A) \to \mathit{TPred} \\ (f \;\doteq\; g)\;t \;=\; f\;t \equiv g\;t \end{array}$$

A temporal predicate can be converted into a *time-invariant* property by requiring it to hold at all points in time:

$$\begin{array}{l} \mathit{Always} \;:\; \mathit{TPred} \to \mathit{Set} \\ \mathit{Always}\;\varphi \;=\; \forall\;t \to \varphi\;t \end{array}$$

6.4 Properties of Time

Depending on the time-domain of a temporal logic, different temporal formulae are valid. This allows some properties of the underlying time domain to be defined at the temporal-logic level by stating temporal logic formulae that hold *if and only if* the time domain has that property [124]. For example, assuming a linear time domain, the properties of time being *dense*, and of having a *first point* and an *end point* can be expressed as follows:

$$\begin{aligned}&\mathit{Density} : \mathit{Set}\\&\mathit{Density} = \forall\, \varphi \rightarrow \mathit{Always}\ (\mathbf{F}\ \varphi \Rightarrow \mathbf{F}\ (\mathbf{F}\ \varphi))\end{aligned}$$

$$\begin{aligned}&\mathit{FirstPoint} : \mathit{Set}\\&\mathit{FirstPoint} = \mathit{Always}\ (\mathbf{P}^{\mathbf{r}}\ (\mathbf{H}\ \bot))\end{aligned}$$

$$\begin{aligned}&\mathit{EndPoint} : \mathit{Set}\\&\mathit{EndPoint} = \mathit{Always}\ (\mathbf{F}^{\mathbf{r}}\ (\mathbf{G}\ \bot))\end{aligned}$$

In the FRP time domain, the *Density* and *FirstPoint* properties hold, whereas the *EndPoint* property does not.

6.5 Properties of N-ary FRP

This section uses the temporal operators to define a number of useful properties of signal vectors and signal functions. These properties are defined in terms of the conceptual model of N-ary FRP from Section 4.1.

6.5.1 Pointwise Sample Equality

In the N-ary FRP model, signal vectors are not simply functions from time to value. Consequently, the time-varying equality from Section 6.3 cannot be used directly to define a pointwise equality of signal vectors. Instead, the *sample* function from Section 5.1 is used to define a pointwise *sample equality*:

$$\begin{aligned}&\mathit{EqSample} : \mathit{SigVec}\ \mathit{as} \rightarrow \mathit{SigVec}\ \mathit{as} \rightarrow \mathit{TPred}\\&\mathit{EqSample}\ s_1\ s_2 = \mathit{sample}\ s_1 \doteq \mathit{sample}\ s_2\end{aligned}$$

Other notions of pointwise signal-vector equality are also possible, as will be discussed in Section 8.3.2.

6.5.2 Causality and Decoupledness

As discussed in Section 3.2.2, all signal functions are required to be *temporally causal*. That is, the output of a signal function at any time t is uniquely determined by its input over the interval $[0, t]$. This can be formulated as follows:

$$\begin{aligned}&\mathit{Causal} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \rightarrow \mathit{Set}\\&\mathit{Causal}\ \mathit{sf} = \forall\, s_1\ s_2 \rightarrow \mathit{Always}\ (\mathbf{H}^{\mathbf{r}}\ (\mathit{EqSample}\ s_1\ s_2) \Rightarrow \mathit{EqSample}\ (\mathit{sf}\ s_1)\ (\mathit{sf}\ s_2))\end{aligned}$$

Intuitively, this says that a signal function *sf* is causal if, for any two signal vectors s_1 and s_2, s_1 and s_2 having been equal *up to* any time point implies that $\mathit{sf}\ s_1$ and $\mathit{sf}\ s_2$ are equal *at* that time point.

A related notion is *temporal decoupledness*. Informally, a signal function is *temporally decoupled* if its output at time t is uniquely determined by its input over the interval $[0, t)$. Crucially, this excludes its input *at* time t. This can be formulated as follows:

$$\begin{aligned}&\mathit{Decoupled} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \rightarrow \mathit{Set}\\&\mathit{Decoupled}\ \mathit{sf} = \forall\, s_1\ s_2 \rightarrow \mathit{Always}\ (\mathbf{H}\ (\mathit{EqSample}\ s_1\ s_2) \Rightarrow \mathit{EqSample}\ (\mathit{sf}\ s_1)\ (\mathit{sf}\ s_2))\end{aligned}$$

The term *decoupled* has many different meanings, but in this thesis *decoupled signal function* will mean a signal function that is temporally decoupled.

As will be discussed in Chapter 7, identifying decoupled signal functions is particularly useful as they can be used to guarantee well-defined feedback. However, for that purpose, a stronger version of decoupledness is also required.

Informally, a signal function is *strictly decoupled* if there exists some time delta ($\delta : \Delta t$) such that the signal function's output at time t is uniquely determined by its input over the interval $[0, t - \delta]$. This can be formulated as follows:

```
StrictlyDec : SF as bs → Set
StrictlyDec sf  =  Σ Δt (λ δ →
   (∀ s₁ s₂ → Always (Earlier δ (Hʳ (EqSample s₁ s₂)) ⇒ EqSample (sf s₁) (sf s₂))))
   where
      Earlier : Δt → TPred → TPred
      Earlier δ φ t  |  t ⩾ δ  =  φ (t − δ)
                     |  t < δ  =  True
```

Note that strict decoupledness implies decoupledness, and decoupledness implies causality:

```
StrictlyDec sf  → Decoupled sf
Decoupled sf    → Causal sf
```

Finally, note that causality, decoupledness and strict decoupledness correspond to the more general notions of *non-expansive*, *weakly contractive* and *strictly contractive* functions, respectively [67].

6.5.3 Statelessness

Recall from Chapter 5 that some signal functions are such that they can be implemented without an internal state. These signal functions correspond to lifted pure functions, and are characterised by their output at any given time point being uniquely determined by their input at that time point. This can be formulated as follows:

```
Stateless : SF as bs → Set
Stateless sf  =  ∀ s₁ s₂ t₁ t₂ → sample s₁ t₁ ≡ sample s₂ t₂ → sample (sf s₁) t₁ ≡ sample (sf s₂) t₂
```

Note that statelessness trivially implies causality:

```
Stateless sf  → Causal sf
```

Signal functions that are not stateless are known as *stateful* signal functions. In other settings, the terms *combinatorial* and *sequential* are used for the same notions as stateless and stateful, respectively.

6.5.4 Properties of Primitives

This sections considers which properties hold for the N-ary FRP primitive signal functions, and which properties are preserved by the N-ary FRP primitive combinators. The accompanying proofs for these properties are not given, but most[1] of them have been formally verified in Agda and the proof scripts are available in the online archive [1].

[1]The properties of *switch* and *freeze* are the main exceptions, though some of the more complicated primitive signal functions still remain to be addressed.

Causality

In N-ary FRP, the atomic routers, the signal functions produced by the lifting functions, and all primitive signal functions are causal. Furthermore, all primitive combinators preserve causality:

$$\begin{array}{lll}
\mathit{Causal}\ \mathit{sf}_1 & \times\ \mathit{Causal}\ \mathit{sf}_2 & \rightarrow \mathit{Causal}\ (\mathit{sf}_1 \ggg \mathit{sf}_2) \\
\mathit{Causal}\ \mathit{sf}_1 & \times\ \mathit{Causal}\ \mathit{sf}_2 & \rightarrow \mathit{Causal}\ (\mathit{sf}_1\ \&\&\&\ \mathit{sf}_2) \\
\mathit{Causal}\ \mathit{sf} & \times\ (\forall\ e \rightarrow \mathit{Causal}\ (f\ e)) & \rightarrow \mathit{Causal}\ (\mathit{switch}\ \mathit{sf}\ f) \\
\mathit{Causal}\ \mathit{sf} & & \rightarrow \mathit{Causal}\ (\mathit{freeze}\ \mathit{sf}) \\
\mathit{Causal}\ \mathit{sf} & & \rightarrow \forall\ s\ t \rightarrow \mathit{Causal}\ (\mathit{frozenSample}\ \mathit{sf}\ s\ t)
\end{array}$$

The final property expresses that if a signal function is causal, then all frozen versions of that signal function will be causal. The utility function *frozenSample* is defined as follows:

$$\begin{array}{l}
\mathit{frozenSample}\ :\ \mathit{SF}\ \mathit{as}\ \mathit{bs} \rightarrow \mathit{SigVec}\ \mathit{as} \rightarrow \mathit{Time} \rightarrow \mathit{SF}\ \mathit{as}\ \mathit{bs} \\
\mathit{frozenSample}\ \mathit{sf}\ s\ t\ =\ \mathit{sample}\ (\mathit{snd}\ (\mathit{freeze}\ \mathit{sf}\ s))\ t
\end{array}$$

Consequently, all signal functions definable in the N-ary FRP language are causal.

Decoupledness

The atomic routers, and the signal functions produced by the lifting primitives, are *not* decoupled. However, the following primitive signal functions are decoupled:

constantS, *never*, *now*, *integralS*, *delayC*, *delayE*, *delayS*, *dfromS*

The primitive combinators all preserve decoupledness:

$$\begin{array}{lll}
\mathit{Decoupled}\ \mathit{sf}_1 & \times\ \mathit{Decoupled}\ \mathit{sf}_2 & \rightarrow \mathit{Decoupled}\ (\mathit{sf}_1 \ggg \mathit{sf}_2) \\
\mathit{Decoupled}\ \mathit{sf}_1 & \times\ \mathit{Decoupled}\ \mathit{sf}_2 & \rightarrow \mathit{Decoupled}\ (\mathit{sf}_1\ \&\&\&\ \mathit{sf}_2) \\
\mathit{Decoupled}\ \mathit{sf} & \times\ (\forall\ e \rightarrow \mathit{Decoupled}\ (f\ e)) & \rightarrow \mathit{Decoupled}\ (\mathit{switch}\ \mathit{sf}\ f) \\
\mathit{Decoupled}\ \mathit{sf} & & \rightarrow \mathit{Decoupled}\ (\mathit{freeze}\ \mathit{sf}) \\
\mathit{Decoupled}\ \mathit{sf} & & \rightarrow \forall\ s\ t \rightarrow \mathit{Decoupled}\ (\mathit{frozenSample}\ \mathit{sf}\ s\ t)
\end{array}$$

In the case of sequential composition, there is also the stronger property that the composite signal function is decoupled if one of the component signal functions is decoupled and the other is causal:

$$\begin{array}{lll}
\mathit{Decoupled}\ \mathit{sf}_1 & \times\ \mathit{Causal}\ \mathit{sf}_2 & \rightarrow \mathit{Decoupled}\ (\mathit{sf}_1 \ggg \mathit{sf}_2) \\
\mathit{Causal}\ \mathit{sf}_1 & \times\ \mathit{Decoupled}\ \mathit{sf}_2 & \rightarrow \mathit{Decoupled}\ (\mathit{sf}_1 \ggg \mathit{sf}_2)
\end{array}$$

Strict Decoupledness

The atomic routers, and the signal functions produced by the lifting primitives, are *not* strictly decoupled. However, the following primitive signal functions are strictly decoupled:

constantS, *never*, *now*, *delayC*, *delayE*, *delayS*

The primitive combinators preserve strict decoupledness is the same manner as decoupledness:

$$\begin{array}{lll}
\mathit{StrictlyDec}\ \mathit{sf}_1 & \times\ \mathit{StrictlyDec}\ \mathit{sf}_2 & \rightarrow \mathit{StrictlyDec}\ (\mathit{sf}_1 \ggg \mathit{sf}_2) \\
\mathit{StrictlyDec}\ \mathit{sf}_1 & \times\ \mathit{StrictlyDec}\ \mathit{sf}_2 & \rightarrow \mathit{StrictlyDec}\ (\mathit{sf}_1\ \&\&\&\ \mathit{sf}_2) \\
\mathit{StrictlyDec}\ \mathit{sf} & \times\ (\forall\ e \rightarrow \mathit{StrictlyDec}\ (f\ e)) & \rightarrow \mathit{StrictlyDec}\ (\mathit{switch}\ \mathit{sf}\ f) \\
\mathit{StrictlyDec}\ \mathit{sf} & & \rightarrow \mathit{StrictlyDec}\ (\mathit{freeze}\ \mathit{sf}) \\
\mathit{StrictlyDec}\ \mathit{sf} & & \rightarrow \forall\ s\ t \rightarrow \mathit{StrictlyDec}\ (\mathit{frozenSample}\ \mathit{sf}\ s\ t) \\
\mathit{StrictlyDec}\ \mathit{sf}_1 & \times\ \mathit{Causal}\ \mathit{sf}_2 & \rightarrow \mathit{StrictlyDec}\ (\mathit{sf}_1 \ggg \mathit{sf}_2) \\
\mathit{Causal}\ \mathit{sf}_1 & \times\ \mathit{StrictlyDec}\ \mathit{sf}_2 & \rightarrow \mathit{StrictlyDec}\ (\mathit{sf}_1 \ggg \mathit{sf}_2)
\end{array}$$

Statelessness

The atomic routers, and the signal functions produced by the lifting primitives, *are* stateless. Additionally, the following primitive signal functions are stateless:

$\mathit{constantS}, \mathit{never}, \mathit{filterE}, \mathit{fromS}$

Statelessness is preserved by the $\ggg$, &&& and *freeze* combinators:

$$\begin{array}{lll}
\mathit{Stateless}\ \mathit{sf}_1 \times \mathit{Stateless}\ \mathit{sf}_2 & \rightarrow & \mathit{Stateless}\ (\mathit{sf}_1 \ggg \mathit{sf}_2) \\
\mathit{Stateless}\ \mathit{sf}_1 \times \mathit{Stateless}\ \mathit{sf}_2 & \rightarrow & \mathit{Stateless}\ (\mathit{sf}_1\ \&\&\&\ \mathit{sf}_2) \\
\mathit{Stateless}\ \mathit{sf} & \rightarrow & \mathit{Stateless}\ (\mathit{freeze}\ \mathit{sf}) \\
\mathit{Stateless}\ \mathit{sf} & \rightarrow & \forall\ s\ t \rightarrow \mathit{Stateless}\ (\mathit{frozenSample}\ \mathit{sf}\ s\ t)
\end{array}$$

The *switch* combinator does not preserve statelessness as it is inherently stateful: its output sample at any time point depends on whether the structural switch has occurred prior to that time point.

Note that $\mathit{Stateless}\ \mathit{sf} \rightarrow \mathit{Stateless}\ (\mathit{freeze}\ \mathit{sf})$ only holds under an assumption of signal function extensionality based on equality of samples: that is, assuming two signal functions are equal if for all input signal vectors their output samples are equal at all time points:

$(\forall\ s \rightarrow \mathit{Always}\ (\mathit{EqSample}\ (\mathit{sf}_1\ s)\ (\mathit{sf}_2\ s))) \rightarrow \mathit{sf}_1 \equiv \mathit{sf}_2$

This is because a frozen signal function has been partially applied to a signal vector up to the time point at which it was frozen. The same signal function frozen at a different time point will have been applied to a different amount of input, and thus the two frozen signal functions will not be intensionally equal. For a signal function to be stateless the output sample (which includes the frozen signal function in this case) must solely depend on the input sample, and thus intensionally *freeze sf* cannot be stateless (as the frozen signal function depends on time). However, because the frozen signal functions are themselves stateless, they ignore those differing past inputs and thus are extensionally equal.

6.6 Conclusions

Temporal logic combinators allow time-varying properties to be expressed simply and concisely. As FRP is based around time-varying entities, such combinators are well suited to formalising FRP-specific properties.

In most FRP variants, causality is not inherent to the semantic model, but is instead stated informally as a necessary side-condition. Here, what it means for a signal function to be causal has been formalised. This allowed the property that all N-ary FRP combinators preserve causality to be formalised, and thus the desired property that all signal functions in N-ary FRP are causal can be stated and proven.

The property of a signal function being stateless was also formalised. In other reactive systems, this property is usually inferred from the implementation of the signal function. Here, this allows the property to be deduced from the semantic definition of a signal function, without reference to an implementation. (Though the property is sufficiently simple that this has not been a cause for concern with other systems.)

Finally, decoupledness and strict decoupledness have been formalised, and the signal functions for which they hold have been identified (and in many cases formally proven). These are non-trivial properties, and will be used in Chapter 7 to guarantee that feedback is well-defined.

Chapter 7

Type-safe Feedback

This chapter adds a feedback combinator to N-ary FRP, allowing for recursive definitions at the reactive level. A concern with feedback combinators is that they may allow diverging programs (ill-defined feedback). To address this, the type system of N-ary FRP is refined such that only well-defined feedback is well-typed, thereby allowing the absence of ill-defined feedback to be guaranteed statically. The Agda and Haskell embeddings from Chapter 5 are then extended accordingly.

While this type system refinement is defined in the context of N-ary FRP, it could also be applied to other FRP variants provided they have a first-class signal-function abstraction. Multi-kinded signals and n-ary signal functions are not prerequisites for this approach.

7.1 Causality Analysis

Feedback is a crucial feature in reactive programming as it allows for recursive definitions at the reactive level, thereby allowing signal functions to be mutually dependent upon one another. In terms of data-flow networks, recursive definitions correspond to cyclic network structures. For example, in the Yampa implementation of Space Invaders [27], aliens, guns and missiles are modelled as signal functions. These game entities are all mutually dependent: the aliens and guns react to each other by moving and firing missiles, and the aliens and missiles can be destroyed if they collide. Thus feedback is required so that each game entity can access the output of the other game entities.

In the context of time-varying signals, there are several notions of feedback to consider:

- *Decoupled feedback*: the current input to a signal function can depend on its past outputs.
- *Instantaneous feedback*: the current input to a signal function can depend on its current output.
- *Causal feedback*: the current input to a signal function can depend on its past and current outputs.
- *Acausal feedback*: the current input to a signal function can depend on its past, current and future outputs.

Reactive languages vary according to which type of feedback they permit. Acausal feedback is usually prohibited, but some reactive languages allow causal feedback while others only allow decoupled feedback. The advantage of restricting feedback to the decoupled variety is that it is always well-defined. This will be discussed further in Section 7.2.1, but intuitively this is because at the time the input is needed, the output on which it depends will already have been computed. On the other hand, instantaneous feedback (which is part of causal feedback) requires a fixed-point computation at the reactive level, which can diverge in many cases. However, instantaneous feedback can also be well-defined and useful [82, 117]; ruling it out altogether prohibits such cases.

Determining if all feedback in a program is well-defined is known as *causality analysis* [30, 82, 117]. Many reactive languages perform causality analysis as a compile-time check, so that ill-defined programs can be rejected statically. This causality check may only permit decoupled feedback (such as in Signal [120], Lucid Synchrone [30], or Real-Time FRP [128]), or it may permit causal feedback by checking that all instantaneous feedback is well-defined (such as in Esterel [117]). On the other hand, some reactive languages permit causal feedback without performing any causality checks (such as Yampa [92], or the (nameless) experimental synchronous data-flow languages defined by Edwards and Lee [35] and Lee and Zheng [71]). Such languages rely on the programmer to ensure that feedback is well-defined, and can deadlock at run-time if this is not the case.

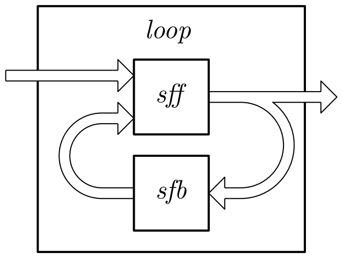
A program can be ensured to contain only decoupled feedback if all fed-back signals pass through a decoupled signal function (defined in Section 6.5.2). Thus, causality analysis for languages that only permit decoupled feedback is a matter of ensuring that a decoupled signal function appears on all feedback paths. This sort of causality analysis is well-studied for static networks [30], but not in the presence of dynamism. For example, the latest version of Lucid Synchrone allows some structural dynamism [17, 22], but at the cost of a very conservative causality analysis: a specific decoupling primitive (a one-time-step delay) must appear *syntactically* on all feedback paths, and it must be in a static part of the network [104].

The N-ary FRP variant described in this chapter, which will be called *N-ary FRP with Feedback*, takes the approach of only permitting decoupled feedback, while avoiding the conservative restrictions of Lucid Synchrone. The basic idea is to encode whether or not a signal function is decoupled within its type. This way, there is not just one specific decoupling primitive: any signal function of the decoupled type can be used. The decoupledness of a composite signal function is computed from that of its components, meaning that user-defined signal functions can also be used for decoupling (a decoupling *primitive* does not have to appear *syntactically* on the feedback path). Furthermore, this allows decoupling to occur within a dynamic part of the network, as the type (if not the value) of a residual signal function is known in advance.

7.2 Feedback Combinators

N-ary FRP with Feedback consists of the primitives of N-ary FRP, plus an additional routing combinator called *loop* that allows feedback to be expressed. This combinator is based on the feedback combinator from the Arrows framework [101] (which is used by Yampa), but will prove slightly more expressive in this setting (see Section 7.2.2).

Figure 7.1 The N-ary FRP feedback combinator



The type and semantics of *loop* are as follows:

$loop\ :\ SF\ (as, cs)\ bs \rightarrow SF\ bs\ cs \rightarrow SF\ as\ bs$
$loop\ sff\ sfb \approx \lambda\ sa \rightarrow fix\ ((\lambda\ sc \rightarrow sff\ (sa, sc)) \circ sfb)$

Intuitively, *loop* takes two signal functions as arguments, which are called the *feed-forward* (*sff*) and *feedback* (*sfb*) signal functions. The feed-forward signal function takes two inputs: the overall input (*sa*), and the output of the feedback signal function (*sc*). The output of the feed-forward signal function is both the input to the feedback signal function and the overall output of the combinator. This is best understood graphically: see Figure 7.1.

Note that the definition of *loop* uses the fixed-point operator *fix*, which is defined as follows:

$fix\ :\ (A \rightarrow A) \rightarrow A$
$fix\ f\ =\ f\ (fix\ f)$

However, *fix* is not a total function: it is only defined when applied to a function that has a unique fixed point; otherwise, it diverges. Consequently, it is possible for the *loop* combinator to be used to construct ill-defined feedback; for example:

$bad\ :\ SF\ as\ bs$
$bad\ =\ loop\ sfSnd\ identity$

The next section discusses constraints that are sufficient to ensure that signal functions constructed using *loop* are total.

7.2.1 Well-Defined Feedback

The *loop* combinator is total if it only applies the *fix* operator to a signal function that has a unique fixed point. The keys to ensuring this are the decoupled (and strictly decoupled) properties of signal functions defined in Section 6.5.2, and Banach's Fixed-Point Theorem [5]:

Banach's Fixed-Point Theorem: Any strictly contractive endofunction has a unique fixed point, and repeated iteration of the function will converge to that fixed point.

Contractive Functions

Formally characterising N-ary FRP's decoupled and strictly decoupled signal functions as contractive and strictly contractive functions remains as future work; here just the intuition is given.

Banach's theorem is defined on metric spaces, which are sets augmented with a notion of real-valued "distance" between elements of that set. A function f is *contractive* if, for any two elements x and y, the distance between $f\,x$ and $f\,y$ is less than the distance between x and y. Being contractive does not guarantee that a function has a fixed point, as the measure of distance is dense (and thus the distance could continue to decrease without converging). However, a *strictly contractive* function has an additional constraint on the reduction of the distance at each iteration, such that the iterated function sequence *will* converge to a fixed point.

In the case of N-ary FRP, distance corresponds to *time*. More specifically, the distance between two signals depends on the first time point at which they differ: the earlier this time point, the greater the distance. A contractive signal function is one that decreases the distance, which is true of any decoupled signal function (as the output signal of a decoupled signal function at any time t can only depend on the input signal before time t). Similarly, any strictly decoupled signal function is strictly contractive, because each iteration reduces the distance by a fixed amount of time. As the time domain has a start point ($time_0$), a strictly decoupled signal function will thus converge after a finite number of iterations.

Furthermore, a *decoupled* signal function operating over discrete-time signals is also strictly contractive [61, 67], provided that the discrete-time signals have a finite number of occurrences in a finite amount of time (as is the case for Event and Step signals in N-ary FRP). This is because a decoupled signal function will always reduce the distance by at least one occurrence, and thus will converge after a finite number of iterations.

Totality of *loop*

The conclusions of the preceding subsection can be formulated as follows:

```
UniqueFixPoint : SF as as → Set
UniqueFixPoint sf = StrictlyDec sf ⊎ (Decoupled sf × DiscreteSV as)
  where
    DiscreteSV : SVDesc → Set
    DiscreteSV (C _)     = False
    DiscreteSV (E _)     = True
    DiscreteSV (S _)     = True
    DiscreteSV (as, bs)  = DiscreteSV as × DiscreteSV bs
```

This property can be used to derive constraints that ensure *loop* is total.

First, consider a simpler combinator (shown in Figure 7.2a):

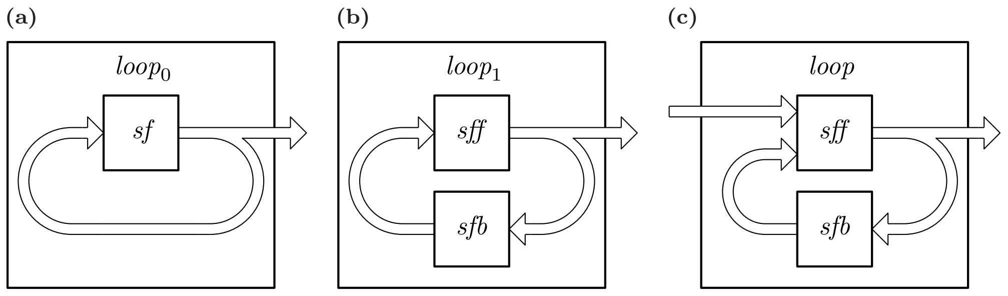
```
loop₀ : SF bs bs → SigVec bs
loop₀ sf = fix sf
```

This combinator is total if either *sf* is strictly decoupled, or *sf* is decoupled and its output contains no Continuous signals. Next consider a variant of this combinator that splits the signal function into two sequentially composed signal functions (Figure 7.2b):

```
loop₁ : SF cs bs → SF bs cs → SigVec bs
loop₁ sff sfb = loop₀ (sfb ⋙ sff)
```

This combinator is total if (*sfb* ⋙ *sff*) has a unique fixed point. As discussed in Section 6.5.4, the sequential composition (in either order) of a decoupled (or strictly decoupled) signal function with a causal signal function is a decoupled (or strictly decoupled) signal function. All signal

Figure 7.2 Deriving the *loop* combinator



functions in N-ary FRP are causal, thus $loop_1$ *sff sfb* is total if any of the following conditions hold:

- *sff* is strictly decoupled;
- *sfb* is strictly decoupled;
- *sff* is decoupled and its output contains no Continuous signals;
- *sfb* is decoupled and its input contains no Continuous signals.

Finally, *loop* (Figure 7.2c) can be defined in terms of $loop_1$:

$$loop : SF\ (as, cs)\ bs \rightarrow SF\ bs\ cs \rightarrow SF\ as\ bs$$
$$loop\ sff\ sfb \approx \lambda\ sa \rightarrow loop_1\ (\lambda\ sc \rightarrow sff\ (sa, sc))\ sfb$$

To determine when this is total, the following lemmas are useful:

$$Decoupled\ sff \ \ \rightarrow Decoupled\ (\lambda\ sc \rightarrow sff\ (sa, sc))$$
$$StrictlyDec\ sff \rightarrow StrictlyDec\ (\lambda\ sc \rightarrow sff\ (sa, sc))$$

Thus, *loop sff sfb* is total if any of the following hold[1]:

- *sff* is strictly decoupled;
- *sfb* is strictly decoupled;
- *sff* is decoupled, and its output contains no Continuous signals;
- *sfb* is decoupled, and its input contains no Continuous signals.

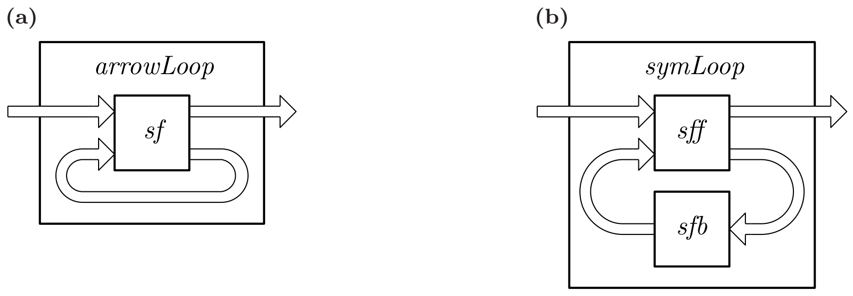
7.2.2 Alternative Feedback Combinators

There are many other possible formulations of a feedback combinator. For the Haskell Arrows framework, Paterson [101] chose a slightly different definition (here renamed *arrowLoop* to avoid confusion). His combinator (shown in Figure 7.3a) can be defined in terms of *loop* as follows:

$$arrowLoop : SF\ (as, cs)\ (bs, cs) \rightarrow SF\ as\ bs$$
$$arrowLoop\ sf = loop\ sf\ sfSnd \ggg sfFst$$

[1]Intuitively, it seems likely that if the *output* (as opposed to the input) of *sfb* does not contain any Continuous signals, then *sff* or *sfb* being decoupled is also sufficient to ensure that *loop sff sfb* is total; but this remains to be proved.

Figure 7.3 Alternative feedback combinators



As *sfSnd* is not decoupled, the *arrowLoop* combinator is total if *sf* is strictly decoupled, or if *sf* is decoupled and *cs* contains no Continuous signals. The reason *arrowLoop* is not used as the feedback primitive in N-ary FRP is that this constraint is more restrictive: it does not, for example, allow the overall output (*bs*) to depend instantaneously on the overall input (*as*), whereas that can be expressed using *loop*.

In previous work [112], I defined another feedback combinator (renamed *symLoop* and shown in Figure 7.3b), which can also be defined in terms of *loop*:

```
symLoop : SF (as, cs) (bs, ds) → SF ds cs → SF as bs
symLoop sff sfb = loop sff (sfSnd ⋙ sfb) ⋙ sfFst
```

Note that the following lemmas hold (by the properties in Section 6.5.4):

```
Decoupled sfb   → Decoupled (sfSnd ⋙ sfb)
StrictlyDec sfb → StrictlyDec (sfSnd ⋙ sfb)
```

Thus, *symLoop* is total if either *sff* or *sfb* is strictly decoupled, or either of them is decoupled and *ds* contains no Continuous signals. The reason *symLoop* is not used as the N-ary FRP feedback primitive is merely that *loop* is simpler. The two are interdefinable:

```
loop : SF (as, cs) bs → SF bs cs → SF as bs
loop sff sfb = symLoop (sff ⋙ sfFork) sfb
```

7.3 Type System for N-ary FRP with Feedback

Having determined sufficient constraints to ensure that feedback is well-defined, the next task is to encode those constraints in the type system as a form of *refinement type* [41]. One approach would be to encode them directly, for example:

```
loop : (sff : SF (as, cs) bs) → (sfb : SF bs cs)
         → StrictlyDec sff ⊎ StrictlyDec sfb
         ⊎ (Decoupled sff × DiscreteSV bs) ⊎ (Decoupled sfb × DiscreteSV bs)
         → SF as bs
```

However, this type is both rather daunting and unsuitable for practical usage. First, while it could be embedded in a dependently typed language such as Agda, it could not be expressed in a language such as Haskell. Second, it places an unpleasant proof burden on the programmer. This section presents an alternative refinement of the type system that is not quite as precise, but is suitable for embedding in Haskell and does not have the same proof burden.

The key idea is to index the *SF* type with the required information [131]. The decoupledness of a signal function is then given by the value of that index, with each combinator computing its index from the indices of its component signal functions.

Also, *loop* is further constrained such that it must be the feedback signal function that is decoupled (or strictly decoupled):

```
loop : (sff : SF (as, cs) bs) → (sfb : SF bs cs) → StrictlyDec sfb ⊎ (Decoupled sfb × DiscreteSV bs)
         → SF as bs
```

This is a little restrictive, but does significantly simplify the type system. Though not done here, the lost expressiveness could be regained by providing an additional feedback combinator that requires the feed-forward signal function to be the decoupled one (see Section 7.3.3).

7.3.1 Decoupledness Indices

First, decoupledness information needs to be represented. This is achieved by a data type of decoupledness values:

```
data Dec : Set where
  cau : Dec  -- causal
  dec : Dec  -- decoupled
```

The signal function type is then indexed by such a value, giving the following refined conceptual definition:

```
SF : SVDesc → SVDesc → Dec → Set
SF as bs cau ≈ {sf ∈ (SigVec as → SigVec bs) | Causal sf}
SF as bs dec ≈ {sf ∈ (SigVec as → SigVec bs) | StrictlyDec sf ⊎ (Decoupled sf × DiscreteSV as)}
```

It is important to note that this does not divide signal functions into two mutually exclusive sets. As previously discussed, all decoupled signal functions are causal, and thus (*SF as bs* dec) is a *subtype* of (*SF as bs* cau). In a host language that provides subtyping, this could be encoded directly. However, in host languages without subtyping, such as Agda and Haskell, explicit coercion is required. For this purpose, the following primitive combinator is added:

```
weaken : SF as bs d → SF as bs cau
weaken sf ≈ sf
```

Finally, note that the type *Dec* is isomorphic to *Bool*. Indeed, Booleans could have been used for this purpose; the *Dec* type was introduced only for clarity. Conjunction and disjunction can thus be overloaded onto *Dec*, equating dec with true and cau with false:

```
_∨_ : Dec → Dec → Dec
dec ∨ d = dec
cau ∨ d = d

_∧_ : Dec → Dec → Dec
dec ∧ d = d
cau ∧ d = cau
```

7.3.2 Refined Primitives

The refined types of the N-ary FRP primitives are given in Listing 7.1. Note that signal functions are indexed dec or cau, and that combinators compute their decoupledness from that of their components using Boolean operators. These values and formulae come directly from the properties in Section 6.5.4.

Listing 7.1 Primitives of N-ary FRP with Feedback

$identity$	$: SF\ as\ as\ \mathsf{cau}$
$sfFst$	$: SF\ (as, bs)\ as\ \mathsf{cau}$
$sfSnd$	$: SF\ (as, bs)\ bs\ \mathsf{cau}$
$\_\ggg\_$	$: SF\ as\ bs\ d_1 \rightarrow SF\ bs\ cs\ d_2 \rightarrow SF\ as\ cs\ (d_1 \vee d_2)$
$\_\&\&\&\_$	$: SF\ as\ bs\ d_1 \rightarrow SF\ as\ cs\ d_2 \rightarrow SF\ as\ (bs, cs)\ (d_1 \wedge d_2)$
$switch$	$: SF\ as\ (bs, \mathsf{E}\ A)\ d_1 \rightarrow (A \rightarrow SF\ as\ bs\ d_2) \rightarrow SF\ as\ bs\ (d_1 \wedge d_2)$
$freeze$	$: SF\ as\ bs\ d \rightarrow SF\ as\ (bs, \mathsf{C}\ (SF\ as\ bs\ d))\ d$
$loop$	$: SF\ (as, cs)\ bs\ d \rightarrow SF\ bs\ cs\ \mathsf{dec} \rightarrow SF\ as\ bs\ d$
$weaken$	$: SF\ as\ bs\ d \rightarrow SF\ as\ bs\ \mathsf{cau}$
$liftC$	$: (A \rightarrow B) \rightarrow SF\ (\mathsf{C}\ A)\ (\mathsf{C}\ B)\ \mathsf{cau}$
$liftS$	$: (A \rightarrow B) \rightarrow SF\ (\mathsf{S}\ A)\ (\mathsf{S}\ B)\ \mathsf{cau}$
$liftE$	$: (A \rightarrow B) \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ B)\ \mathsf{cau}$
$liftC2$	$: (A \rightarrow B \rightarrow Z) \rightarrow SF\ (\mathsf{C}\ A, \mathsf{C}\ B)\ (\mathsf{C}\ Z)\ \mathsf{cau}$
$liftS2$	$: (A \rightarrow B \rightarrow Z) \rightarrow SF\ (\mathsf{S}\ A, \mathsf{S}\ B)\ (\mathsf{S}\ Z)\ \mathsf{cau}$
$merge$	$: (A \rightarrow Z) \rightarrow (B \rightarrow Z) \rightarrow (A \rightarrow B \rightarrow Z) \rightarrow SF\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\ \mathsf{cau}$
$join$	$: (A \rightarrow B \rightarrow Z) \rightarrow SF\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\ \mathsf{cau}$
$sampleWithC$	$: (A \rightarrow B \rightarrow Z) \rightarrow SF\ (\mathsf{C}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\ \mathsf{cau}$
$sampleWithS$	$: (A \rightarrow B \rightarrow Z) \rightarrow SF\ (\mathsf{S}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\ \mathsf{cau}$
$constantS$	$: A \rightarrow SF\ as\ (\mathsf{S}\ A)\ \mathsf{dec}$
$never$	$: SF\ as\ (\mathsf{E}\ A)\ \mathsf{dec}$
now	$: SF\ as\ (\mathsf{E}\ Unit)\ \mathsf{dec}$
$notYet$	$: SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)\ \mathsf{cau}$
$filterE$	$: (A \rightarrow Bool) \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)\ \mathsf{cau}$
$hold$	$: A \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{S}\ A)\ \mathsf{cau}$
$edge$	$: SF\ (\mathsf{S}\ Bool)\ (\mathsf{E}\ Unit)\ \mathsf{cau}$
$when$	$: (A \rightarrow Bool) \rightarrow SF\ (\mathsf{C}\ A)\ (\mathsf{E}\ A)\ \mathsf{cau}$
$integralS$	$: SF\ (\mathsf{S}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})\ \mathsf{dec}$
$integralC$	$: SF\ (\mathsf{C}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R})\ \mathsf{cau}$
$fromS$	$: SF\ (\mathsf{S}\ A)\ (\mathsf{C}\ A)\ \mathsf{cau}$
$dfromS$	$: A \rightarrow SF\ (\mathsf{S}\ A)\ (\mathsf{C}\ A)\ \mathsf{dec}$
$delayC$	$: Time^{+} \rightarrow (Time \rightarrow A) \rightarrow SF\ (\mathsf{C}\ A)\ (\mathsf{C}\ A)\ \mathsf{dec}$
$delayS$	$: Time^{+} \rightarrow A \rightarrow SF\ (\mathsf{S}\ A)\ (\mathsf{S}\ A)\ \mathsf{dec}$
$delayE$	$: Time^{+} \rightarrow SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)\ \mathsf{dec}$

The interesting case is of course *loop*:

$$loop\ :\ SF\ (as, cs)\ bs\ d \rightarrow SF\ bs\ cs\ \mathsf{dec} \rightarrow SF\ as\ bs\ d$$

The feedback signal function is required to be decoupled, thereby ensuring that only well-defined feedback is well-typed.

Note that *integralC* is typed as causal. However, there are many different means for an implementation to compute an integral; some decoupled, some not. In practice, reactive languages that are concerned with numerical accuracy provide several integration methods and let the programmer choose the one most suitable for the task at hand. Thus, in some instances, *integralC* will be decoupled.

7.3.3 An Additional Feedback Combinator

As previously mentioned, the type of loop is more constrained than it needs to be, as it excludes the case where the feed-forward signal function is the decoupled one. The more general version would have the following type:

$$generalLoop\ :\ SF\ (as, cs)\ bs\ d_1 \rightarrow SF\ bs\ cs\ d_2 \rightarrow d_1 \vee d_2 \equiv \mathsf{dec} \rightarrow SF\ as\ bs\ d_1$$

However, this uses a data type of propositional equality, which, while embeddable in some host-languages [45, 73], unnecessarily complicates the type system. A simpler option would be to provide an additional feedback combinator that requires the *feed-forward* signal function to be decoupled, for example:

$loop' \;:\; SF\;(as, cs)\;bs\;\mathsf{dec} \to SF\;bs\;cs\;d \to SF\;as\;bs\;\mathsf{dec}$

It is almost possible to express such a combinator in terms of *loop*. However, the type system isn't precise enough to recognise that the resultant signal function is decoupled, typing it as causal instead:

$loop' \;:\; SF\;(as, cs)\;bs\;\mathsf{dec} \to SF\;bs\;cs\;d \to SF\;as\;bs\;\mathsf{cau}$
$loop'\;sff\;sfb \;=\; loop\;(sfSecond\;(forkFirst\;sfb) \ggg sfAssocL)\;(sfFst \ggg sff) \ggg sfSnd$

In general, the type system of N-ary FRP with Feedback is conservative in that some information about which output signals are temporally decoupled from which input signals is lost when using combinators. This lack of precision will be addressed in Section 9.3.

7.4 Feedback Example

As previously mentioned, the FRVR project [12] exploits the ability to freeze signal functions as a means of saving and resuming the world state. Feedback is also essential to that functionality. This section demonstrates how such functionality could be encoded, serving both as a non-trivial example of feedback, and as an example of a situation where the type system prevents instantaneous feedback from being inadvertently defined.

7.4.1 Saving and Resuming

Intuitively, the save-and-resume behaviour is achieved by freezing the signal function that needs to be saved, feeding it back as an additional input, then switching it in when resumption is required. This is best broken down into several combinators.

First, consider a combinator that *saves* the state of a signal function whenever an input event is received. Essentially, this is simply a matter of sampling the output of *freeze*:

$save \;:\; SF\;as\;bs\;d \to SF\;(as, \mathsf{E}\;A)\;(bs, \mathsf{E}\;(SF\;as\;bs\;d))\;\mathsf{cau}$
$save\;sf \;=\; sfFirst\;(freeze\;sf) \ggg sfAssocR \ggg sfSecond\;sampleC$

Next, consider the following combinator that replaces the subordinate signal while maintaining the saving behaviour (using the *replace* combinator from Section 4.3.1):

$saveReplace \;:\; SF\;as\;bs\;d \to SF\;((as, \mathsf{E}\;A), \mathsf{E}\;(SF\;as\;bs\;d))\;(bs, \mathsf{E}\;(SF\;as\;bs\;d))\;\mathsf{cau}$
$saveReplace\;sf \;=\; replace\;(save\;sf)\;save$

The first input event signal ($\mathsf{E}\;A$) controls when saving occurs; the second input event signal ($\mathsf{E}\;(SF\;as\;bs\;d)$) controls when replacing occurs (and carries the replacement signal function).

Finally, the desired saving and resuming behaviour is achieved by closing the feedback loop such that the replacement signal function is taken from the most recently saved signal function. This is achieved by feeding back the saved signal function, storing it in a holding signal function, and sampling that held signal function with the resume event ($\mathsf{E}\;B$):

$saveResume \;:\; SF\;as\;bs\;d \to SF\;((as, \mathsf{E}\;A), \mathsf{E}\;B)\;bs\;\mathsf{cau}$
$saveResume\;sf \;=\; symLoop\;(sfAssocR \ggg sfSecond\;(sfSwap \ggg sampleC) \ggg saveReplace\;sf)\;(dhold\;sf)$

Note the use of *dhold* rather than *hold*. Had *hold* been used, this would have created ill-defined instantaneous feedback. However, the type of *symLoop* is as follows:

$symLoop\ :\ SF\ (as, cs)\ (bs, ds)\ d \rightarrow SF\ ds\ cs\ \mathsf{dec} \rightarrow SF\ as\ bs\ d$

Using *hold* (which is causal) would therefore be type incorrect, and thus the type system prevents this mistake.

7.4.2 Hypothetical Syntax

The point-free style of programming makes the definitions of *save* and *saveResume* somewhat hard to follow. As discussed in Section 3.4.6, an implementation would be expected to provide more convenient syntax. For example, in (hypothetical) arrow-like notation, the *save* and *saveResume* functions might be expressed as follows:

```
save : SF as bs d → SF (as, E A) (bs, E (SF as bs d)) cau
save sf  =  proc (sa, se) → do
                (sb, ssf) ← freeze sf ⤙ sa
                se'       ← sampleC   ⤙ (ssf, se)
                identity              ⤙ (sb, se')

saveResume : SF as bs d → SF ((as, E A), E B) bs cau
saveResume sf  =  proc (sae, se) → do
                      se'         ← sampleC          ⤙ (ssf, se)
                      (sb, sesf)  ← saveReplace sf   ⤙ (sae, se')
                      ssf         ← dhold sf         ⤙ sesf
                      identity                       ⤙ sb
```

7.5 Extending the Agda Embedding

This section extends the Agda embedding from Chapter 5 to N-ary FRP with Feedback. As previously noted, Agda performs totality and termination checks. Thus, the implementation is confirmed not to allow ill-defined feedback.

Much of the extended embedding is unchanged (or has only trivial changes) from that in Chapter 5. Consequently, only the noteworthy modifications are discussed in this section; the complete code can be found in Appendix C.3.

7.5.1 A Decoupled Transition Function

The core of the implementation in Chapter 5 was a transition function with the following type:

$\Delta t \rightarrow Q \rightarrow Sample\ as \rightarrow Q \times Sample\ bs$

This will be referred to as a *causal transition function*.

Recall that in this style of sampled implementation, the state (Q) is a means of recording any required information about past inputs. For a decoupled signal function, the output sample cannot depend on the current input sample, but it can depend on past input samples (the internal state). However, the updated state *can* depend upon the current input sample. This suggests that an implementation of decoupled signal functions should be built around a *decoupled transition function* of the following type:

$\Delta t \rightarrow Q \rightarrow (Sample\ as \rightarrow Q) \times Sample\ bs$

This is similar to the causal transition function, except that the output sample no longer depends on the input sample. Variants of this decoupled transition function will be ubiquitous in the following implementation.

7.5.2 Nodes

Primitive signal functions may or may not be decoupled. The *Node* data type, which represents primitive signal functions, is thus extended with a decoupled case:

```
data Node (as bs : SVDesc) : Dec → Set where
  cnode : ∀ {Q} → (Δt → Q → Sample as → Q × Sample bs)   → Q → Node as bs cau
  dnode : ∀ {Q} → (Δt → Q → (Sample as → Q) × Sample bs) → Q → Node as bs dec
```

Observe that the cnode constructor contains a causal transition function and is indexed cau, whereas the dnode constructor contains a decoupled signal function and is indexed dec.

The *stepNode* function is then modified accordingly:

```
stepNode : ∀ {as bs d} → Δt → Node as bs d → Sample as → Node as bs d × Sample bs
stepNode δ (cnode f q) sa = first (cnode f) (f δ q sa)
stepNode δ (dnode f q) sa = first (λ g → dnode f (g sa)) (f δ q)
```

The *stepNode* function is itself a causal transition function. A decoupled version of *stepNode* can be defined, but only for decoupled nodes:

```
dstepNode : ∀ {as bs} → Δt → Node as bs dec → (Sample as → Node as bs dec) × Sample bs
dstepNode δ (dnode f q) = first (λ g sa → dnode f (g sa)) (f δ q)
```

Agda accepts this function as total because the type index dec does not match that of the cnode constructor; that is, there cannot be a cnode case. Without the type indices, this function would be partial.

7.5.3 Signal Functions

Modifying the signal-function data types (from Listing 5.2) is fairly straightforward (shown in Listing 7.2). The main differences are the addition of decoupledness indices, and the new constructors for the *loop* and *weaken* combinators. Also, in *SF* the prim constructor is split into two. This is because, in the unmodified *SF* data type, prim contains a *causal* transition function (to apply in the initialisation step), and thus an additional constructor with a *decoupled* transition function is needed (so that the initial output sample of a decoupled signal function does not depend on its initial input sample). This is not required for *SF′*, as its prim constructor does not contain a transition function.

Modifying the step functions requires more work. Only $step_0$ is discussed, as the modifications to *step′* are essentially the same. The cases for the new constructors are as follows:

```
step₀ : ∀ {d as bs} → SF as bs d → Sample as → SF′ as bs d × Sample bs
step₀ (dprim f sb) sa = (prim (f sa), sb)
step₀ (weakener sf) sa = first weakener (step₀ sf sa)
step₀ (looper sff sfb) sa with dstep₀ sfb
... | (g, sc) with step₀ sff (sa, sc)
... | (sff′, sb) = (looper sff′ (g sb), sb)
```

The dprim and weakener cases are straightforward. The interesting case is looper, as one would expect given that this is where the feedback happens. Intuitively, the situation is that both

Listing 7.2 Indexed signal functions

```
data SF  : SVDesc → SVDesc → Dec → Set where
  cprim     : ∀ {as bs} → (Sample as → Node as bs cau × Sample bs)  → SF as bs cau
  dprim     : ∀ {as bs} → (Sample as → Node as bs dec) → Sample bs → SF as bs dec
  arouter   : ∀ {as bs} → AtomicRouter as bs                         → SF as bs cau
  seq       : ∀ {d1 d2 as bs cs} → SF as bs d1 → SF bs cs d2         → SF as cs (d1 ∨ d2)
  fan       : ∀ {d1 d2 as bs cs} → SF as bs d1 → SF as cs d2         → SF as (bs, cs) (d1 ∧ d2)
  rswitcher : ∀ {d1 d2 as bs A} → SF as (bs, E A) d1 → (A → SF as (bs, E A) d2) → SF as bs (d1 ∧ d2)
  freezer   : ∀ {d as bs} → SF as bs d                               → SF as (bs, C (SF as bs d)) d
  looper    : ∀ {d as bs cs} → SF (as, cs) bs d → SF bs cs dec       → SF as bs d
  weakener  : ∀ {d as bs} → SF as bs d                               → SF as bs cau

data SF′  : SVDesc → SVDesc → Dec → Set where
  prim      : ∀ {d as bs} → Node as bs d                        → SF′ as bs d
  arouter   : ∀ {as bs} → AtomicRouter as bs                    → SF′ as bs cau
  seq       : ∀ {d1 d2 as bs cs} → SF′ as bs d1 → SF′ bs cs d2  → SF′ as cs (d1 ∨ d2)
  fan       : ∀ {d1 d2 as bs cs} → SF′ as bs d1 → SF′ as cs d2  → SF′ as (bs, cs) (d1 ∧ d2)
  rswitcher : ∀ {d1 d2 as bs A} → SF′ as (bs, E A) d1 → (A → SF as (bs, E A) d2) → SF′ as bs (d1 ∧ d2)
  freezer   : ∀ {d as bs} → SF′ as bs d                         → SF′ as (bs, C (SF as bs d)) d
  looper    : ∀ {d as bs cs} → SF′ (as, cs) bs d → SF′ bs cs dec → SF′ as bs d
  weakener  : ∀ {d as bs} → SF′ as bs d                         → SF′ as bs cau
```

component signal functions require input from the other, and thus there is no natural order of execution. This is addressed by using $dstep_0$, a decoupled version of $step_0$ (see Listing 7.3). Because the feedback signal function is decoupled, $dstep_0$ can extract the output of the feedback signal function before providing its input. Thus the order of execution is: extract the output from the feedback signal function; execute the feed-forward signal function; provide the input to the feedback signal function. The crucial point is that this only works because the feedback signal function is guaranteed to be decoupled; otherwise $dstep_0$ couldn't be applied to it.

Next consider the $dstep_0$ function itself (Listing 7.3). For unimportant technical reasons, it is necessary to introduce an auxiliary function ($dstepAux_0$) that takes a proof that the signal function has a dec index. There are two key points about $dstepAux_0$. First, the cases for any causal signal function can be rendered absurd by pattern matching on the proof. Thus only the decoupled cases need to be considered. Second, some of the combinators pattern match on the decoupledness indices of their component signal functions to determine whether they are decoupled. Once a component signal function is determined to be decoupled, $dstep_0$ can be recursively applied to it.

The case for seq is the most complicated, so that will be considered as an example. First the index of sf_1 is pattern matched on. If that index is dec, then sf_1 can be executed using $dstep_0$. The output sample (sb) can then be used as input to sf_2, which is executed using $step_0$ to produce the overall output sample (sc). Finally, a function is constructed that, given an input sample (sa), will produce the updated signal function. On the other hand, if sf_1 has a cau index, then sf_2 must be decoupled (otherwise the composite signal function would not be). Thus, sf_2 can be executed using $dstep_0$, producing the output sample (sc). Execution of sf_1 is deferred by placing it within the update function, such that when the input sample (sa) becomes available, sf_1 is executed using $step_0$, and then the updated signal function is constructed.

Listing 7.3 A decoupled step function

```
dstep0 : ∀ {as bs} → SF as bs dec → (Sample as → SF′ as bs dec) × Sample bs
dstep0 sf = dstepAux0 sf refl

dstepAux0 : ∀ {d as bs} → SF as bs d → d ≡ dec → (Sample as → SF′ as bs dec) × Sample bs

dstepAux0 (cprim f) ()

dstepAux0 (dprim f sb) refl = (prim ∘ f, sb)

dstepAux0 (arouter r) ()

dstepAux0 (seq {dec} sf1 sf2) refl with dstep0 sf1
... | (g, sb) with step0 sf2 sb
... | (sf′2, sc) = ((λ sa → seq (g sa) sf′2), sc)
dstepAux0 (seq {cau} {.dec} {as} {bs} {cs} sf1 sf2) refl with dstep0 sf2
... | (g, sc) = (aux, sc)
  where aux : Sample as → SF′ as cs dec
        aux sa with step0 sf1 sa
        ... | (sf′1, sb) = seq sf′1 (g sb)

dstepAux0 (fan {cau} sf1 sf2) ()
dstepAux0 (fan {dec} sf1 sf2) refl with dstep0 sf1 | dstep0 sf2
... | (g1, sb) | (g2, sc) = ((λ sa → fan (g1 sa) (g2 sa)), (sb, sc))

dstepAux0 (rswitcher {cau} sf f) ()
dstepAux0 (rswitcher {dec} sf f) refl with dstep0 sf
... | (g, (sb, nothing)) = ((λ sa → rswitcher (g sa) f), sb)
... | (_, (_, just e)) with dstep0 (f e)
... | (g, (sb, _)) = ((λ sa → rswitcher (g sa) f), sb)

dstepAux0 (freezer sf) refl with dstep0 sf
... | (g, sb) = (freezer ∘ g, (sb, sf))

dstepAux0 (looper sff sfb) refl with dstep0 sff
... | (g, sb) with step0 sfb sb
... | (sfb′, sc) = ((λ sa → looper (g (sa, sc)) sfb′), sb)

dstepAux0 (weakener sf) ()
```

7.5.4 Constructing Primitives

The construction functions from Section 5.2.4 need to be modified to account for decoupledness. First, a function to construct a general decoupled signal function is defined as follows:

```
mkSFdec : ∀ {as bs Q} → (Δt → Q → (Sample as → Q) × Sample bs) → (Sample as → Q)
            → Sample bs → SF as bs dec
mkSFdec f g = dprim (dnode f ∘ g)
```

The *mkSFtimeless* and *mkSFstateless* functions produce causal signal functions, so are essentially the same as before. However, as *mkSFsource* and *mkSFchangeless* produce decoupled signal functions, they should be defined in terms of *mkSFdec*:

```
mkSFsource : ∀ {as bs Q} → (Δt → Q → Q × Sample bs) → Q → Sample bs → SF as bs dec
mkSFsource f q = mkSFdec ((result2 ∘ first) const f) (const q)

mkSFchangeless : ∀ {as bs} → Sample bs → SF as bs dec
mkSFchangeless sb = mkSFsource (λ _ _ → (unit, sb)) unit sb
```

Note that *mkSFchangeless* is now defined in terms of *mkSFsource*, as defining it in terms of *mkSFstateless* would hide that it constructs decoupled signal functions. Having done this, most of the primitive signal function definitions do not require any modification. The exceptions are *dfromS*, *integralC*, and the *delay* family, which are now defined using *mkSFdec*. For example:

```
dfromS : ∀ {A} → A → SF (S A) (C A) dec
dfromS = mkSFdec (λ _ q → (id, q)) id
```

7.5.5 Drawbacks of the Agda Embedding

A drawback of this particular approach comes from the way type-level Booleans (the *Dec* type) have been embedded. The logical operators are defined by pattern matching on their first argument, which means that the first argument has to be known in order for the operator to β-reduce. Much of the time the first argument *is* known, and so this is not a problem, but sometimes when defining new combinators the first argument is not known. Consider, for example, the combinators *sfFirst* and *sfSecond*. There is no problem when defining *sfSecond*, as the index of *identity* is known:

```
sfSecond : ∀ {d as bs cs} → SF bs cs d → SF (as, bs) (as, cs) cau
sfSecond sf = identity *** sf
```

The index of *sfSecond* is computed to be (cau $\wedge$ d), which β-reduces to cau.

However, in the case of *sfFirst*, the index is computed to be (d $\wedge$ cau), which does not β-reduce. This can be dealt with in numerous ways in Agda, the simplest of which is to pattern match on the *Dec* index when defining the combinator. For example:

```
sfFirst : ∀ {d as bs cs} → SF as bs d → SF (as, cs) (bs, cs) cau
sfFirst {cau} sf = sf *** identity
sfFirst {dec} sf = sf *** identity
```

While not hindering the expressiveness of N-ary FRP with Feedback, this is nonetheless a blemish in the embedding of the type system. This will be discussed further in Section 7.7.

7.6 Extending the Haskell Embedding

This section extends the Haskell embedding from Chapter 5 to N-ary FRP with Feedback. Most of this corresponds closely with the extensions to the Agda embedding in the previous section, so only noteworthy differences are discussed. The remainder of the source code can be found in Appendix C.4

7.6.1 Decoupledness Indices

As with signal vector descriptors, decoupledness indices are represented as empty data types:

```
data Dec :: *
data Cau :: *
```

The required Boolean operations over these indices are defined using type families. To allow these to be written in infix form, the *Type Operators* GHC option is required:

```
{-# LANGUAGE TypeOperators #-}
type family   d1  ∨ d2 ::  *
type instance Cau ∨ d2 =  d2
type instance Dec ∨ d2 = Dec
type family   d1  ∧ d2 ::  *
type instance Cau ∧ d2 = Cau
type instance Dec ∧ d2 =  d2
```

Recall that the step function in the Agda embedding pattern matches on decoupledness indices. This cannot be done here: the indices are encoded as types, and Haskell does not permit pattern matching on types.

This difficulty can be overcome by using the technique of *representation types* (also known as *singleton types*) [88, 116]. The basic idea is to reflect the type at the data level, and then pattern match on that data. This is achieved by defining a type-indexed GADT, with a constructor for each type of interest. Pattern matching on the GADT then has the effect of pattern matching on the type index.

A representation type for decoupledness is defined as follows:

```
data DRep :: * → * where
  Dec :: DRep Dec
  Cau :: DRep Cau
```

Pattern matching requires an element of the type to match on. Such an element can be acquired by defining a *Decoupled* type class that provides a function mapping a signal function to a representation of its decoupledness index:

```
class Decoupled d where
  drep :: SF as bs d → DRep d

instance Decoupled Cau where
  drep _ = Cau
instance Decoupled Dec where
  drep _ = Dec
```

Note that *drep* does not use the *value* of the signal function: the *DRep* is computed entirely from its *type*.

Whenever it is necessary to pattern match on the index of a signal function, *drep* can produce a representation of that index, and then that representation can be pattern matched on. For example, the case for sequential composition in the *dstep0* function is encoded as follows:

```
dstep0 (Seq sf1 sf2) = case drep sf1 of
                          Dec → let (g, sb)    = dstep0 sf1
                                    (sf2′, sc) = step0 sf2 sb
                                in ((λsa → Seq′ (g sa) sf2′), sc)
                          Cau → let (g, sc) = dstep0 sf2
                                in (λsa → let (sf1′, sb) = step0 sf1 sa
                                          in Seq′ sf1′ (g sb)
                                   , sc)
```

Because of this pattern matching technique, the definition can proceed in essentially the same manner as its Agda counterpart (see Listing 7.3).

7.6.2 Drawbacks of the Haskell Embedding

The remainder of the implementation is sufficiently similar to the Agda embedding that the code is relegated to Appendix C.4. However, it is also similar enough to suffer from the same blemish discussed in Section 7.5.5: the decoupledness indices do not always β-reduce when defining new combinators. There are fewer ways of dealing with this in Haskell, but the technique of pattern matching on the index is sufficient. For example, *sfFirst* is defined as follows:

```
sfFirst :: Decoupled d ⇒ SF as bs d → SF (as, cs) (bs, cs) Cau
sfFirst sf = case drep sf of
               Cau → sf *** identity
               Dec → sf *** identity
```

7.7 Conclusions

This chapter defined an FRP variant called N-ary FRP with Feedback, which consists of N-ary FRP extended with a feedback combinator and a refined type system. The type system refinement ensures that only well-defined feedback is well-typed. The Agda and Haskell embeddings of N-ary FRP were then extended to N-ary FRP with Feedback.

As discussed in Section 7.5.5, the main disadvantage of these embeddings is that the host-language type checker does not always solve basic decoupledness constraints. This was dealt with by pattern matching on the decoupledness index, but for practical programming this would quickly become tiresome. In Agda there are easier methods of dealing with this, such as using explicit equality proofs to perform substitutions [13]. This is not possible in Haskell as yet, though there have been proposals for extending Haskell with such functionality [45]. In any case, this still requires explicit reasoning about Boolean expressions, which is annoying when the procedure is automateable.

Ideally, one wants functionality such as that provided by Dependent ML [130], where basic constraints for a limited set of built-in type indices are solved by the type checker. By using the built-in Booleans of such a language, the decoupledness constraints would be resolved with no extra effort required by the FRP programmer. Embedding N-ary FRP with Feedback in Dependent ML would be an interesting avenue for future work.

Finally, note that if N-ary FRP with Feedback was implemented as a stand-along language, then the type-checker could of course include a Boolean constraint solver and likewise avoid the problem.

Chapter 8

Change and Optimisation

How to efficiently implement first-order synchronous data-flow networks with static structure is well-studied [49, 70, 106]. However, dynamism and higher-order data-flow in combination with support for hybrid systems raise new implementation challenges. FRP implementations usually adopt either a *push* or *pull* driven implementation strategy [37, 62]. The essence of the push-driven approach is to react to events as they occur by pushing changes through the system. This is a good fit for discrete-time signals. Pull is the opposite, where the need to compute the current value of a signal necessitates computing the current values of all signals it depends on, thus pulling data through the network. This is a good fit for continuous-time signals. Thus push and pull have complementary strengths, but combining both approaches in one system is challenging.

This chapter contributes to FRP implementation by studying the notions of signal change and how change propagates in a highly structurally dynamic signal processing network, and thus identifying when computation is unnecessary and could be avoided. Hopefully this will help reconcile the advantages of push and pull. Note that structural dynamism, and that signals can change just because time passes, make the problem significantly more complex than standard change propagation in a network with a static structure.

These change properties are studied in the setting of N-ary FRP, which takes *signal functions* as the primary reactive abstraction. Many other approaches to FRP instead make *signals* their primary notion. However, note that a setting of signal functions is also the *natural* choice for studying change and change propagation in a signal processing network: the nodes of the network *are* signal functions in the sense discussed in this thesis, regardless of the surface syntax used to set up the network. Thus much of this study is relevant to FRP in general, not just to FRP versions based on signal functions.

8.1 FRP Optimisation

In order to be reactive (delivering timely responses to external stimuli), an FRP implementation must be discretely sampled. Consequently, if notionally continuous-time signals are provided, a concrete implementation can only approximate the ideal semantics.

However, the aim is to make the approximation as faithful as possible. Here, the semantic distinction between different kinds of signals helps. As discussed in Section 4.1.1, certain uses of signals can be statically ruled-out by making the kinds manifest in the type system. This allows employing an implementation strategy that is appropriate for a specific kind of signal, but which would have risked breaking the abstractions had said uses not been ruled out. Moreover, this also opens up opportunities for signal-kind–specific optimisations.

This section briefly reviews the two basic FRP implementation strategies, before discussing the archetypal optimisation opportunities in an FRP system. This provides background and motivation for the change properties in the next section.

8.1.1 Basic FRP Implementation Strategies

An FRP instance typically employs either a *pull-based* (demand-driven) or *push-based* (data-driven) implementation approach [37, 62].

A pull-based approach repeatedly samples the output signals over a sequence of time steps, recomputing every signal at each step. This is a good approach for signals that change often, as is common for continuous-time signals. In fact, the more frequent the changes, the more efficient this approach. However, a signal that changes only rarely has its value unnecessarily recomputed repeatedly. This is inefficient and scales poorly, as the total amount of computation is proportional to the number of signals, not to the signal activity.

In contrast, a push-based approach only recomputes a signal when a signal it depends on changes. This is a natural fit for discrete-time signals: when nothing changes, no updates are needed. However, in FRP there are signals that depend on time (and thus can change even if the signals they depend on do not), as well as signals that (conceptually) change continuously. The former implies that only reacting to external events is not enough. The second, that there can be a substantial overhead for using an exclusively push-based approach, as a continuous signal would trigger a recomputation at every time step.

Ideally, one would like to employ both strategies selectively, to reap the benefits of each. Developing an understanding of how change works in a setting of dynamic hybrid signal function networks is a step in that direction.

8.1.2 Optimisation Opportunities

For most networks, many signals will be unchanging for significant periods of time, with changes occurring sparsely compared with the sampling rate. Also, given the dynamic nature of FRP and the combinator style used to construct networks, it is very common for:

- signals to be used for a while, but then later ignored;
- signals to change for a while, before becoming constant (consider *inelasticBall* from Section 3.3.4).

An implementation should optimise as much as possible based on these observations, without breaking any abstractions.

The optimisations considered in this chapter can be divided into two archetypes: *change propagation* and *structural optimisation*. Change propagation involves identifying which signal functions are such that their output will not change unless their input does. When executing such signal functions, there is no need to recompute their output while their input does not change. Structural optimisation involves simplifying the network structure by eliminating unnecessary signal functions and combinators, with the aim of making future execution more efficient. To recognise when and where such optimisations are valid, unchanging and unused signals need to be tracked. To this end, it is necessary to determine which signal functions produce unchanging signals, and which signal functions do not use their inputs.

By distinguishing between the three signal kinds, these properties of signals and signal functions can be more precisely tracked, hence making it possible to apply more precise optimisations. For example, Continuous signals are likely to be always changing, no matter how rapid the sampling rate. On the other hand, Step signals tend to change only sparsely, and are thus likely to benefit greatly from change-propagation optimisations. By combining this knowledge with knowledge about how signal functions are affected by change, it becomes possible to select appropriate implementation and optimisation strategies in a fine-grained manner.

8.2 Measuring Efficiency

A reactive system is discretely sampled during execution, regardless of whether the individual samples are triggered by "pushes" or "pulls". In the former case the samples will be at irregular intervals, whereas in the latter case they may be at either regular or irregular intervals. In the following discussion, references to the sampling rate will mean the frequency of sampling if regularly sampled, or the *average* frequency if irregularly sampled.

Many reactive applications either specify some requirement of the sampling rate of the system (such as a minimum or desired rate), or consider the quality of the system performance to be proportional to the sampling rate. Typically, push-based systems require the sampling rate to be at-least the rate of internal "push" events, otherwise a backlog of events will build up. On the other hand, pull-based systems often have a desired rate of sampling, with performance degrading as the sampling rate falls, and perhaps also a minimum sampling rate beyond which the system's performance is unacceptable. Consequently, one good measure of the efficiency of an implementation of a reactive language is the maximum sampling rate it can support. Of course, the maximum sampling rate of any given implementation will vary depending on the complexity of the application: an implementation that can sample rapidly enough for one small application may not be able to sample rapidly enough for another larger application. This makes it hard to measure an implementation's absolute efficiency in this way, but it does allow the relative efficiency of two different implementations to be approximated by comparing their maximum sampling rates for some benchmark applications. There are of course other measures of efficiency that could be considered, such as space usage or reaction latency (the time between the system receiving an input and a corresponding output sample being produced). In this chapter, "efficiency" refers primarily to the rate of sampling, with the aim being to identify optimisations that increase the maximum sampling rate. Note however that many optimisations that reduce the sampling rate also reduce space usage and reaction latency.

Comparing the relative efficiency of existing reactive languages, implementations, and their optimisations, is beyond the scope of this thesis. However, the following two test cases should give a rough idea of the scale of existing sampling rates. These results were obtained on a Dual Core 2Ghz Intel laptop running Ubuntu 10.10, and the code was compiled using GHC 6.12 with optimisations switched on. In both cases, the implementations were set to sample at the maximum rate possible.

The first test case consisted of 50 copies of the *elasticBall* signal function (Section 4.3.2) running in parallel (each with a different initial configuration). The Haskell embedding of N-ary FRP described in Section 5.3 achieved an average sampling rate of 280 samples per second. As a comparison, the latest version of Yampa achieved an average sampling rate of 5100 samples per second. This difference is unsurprising, as the Yampa implementation incorporates a significant amount of optimisation [90].

The second test case uses the *playNotes* signal function, taken from the Yampa Synthesiser [43, Section 3]. This signal function converts a sequence of MIDI note numbers into audio signals, immediately ending a note when the next input note is received (thus it is only ever computing a single note at a time). The test consisted of running 50 copies of *playNotes* in parallel; that is, computing 50 notes simultaneously. The Haskell embedding of N-ary FRP achieved an average sampling rate of 300 samples per second, while Yampa achieved 4100 samples per second.

Further experimentation varying the number of balls (or notes) in each test case suggested that Yampa typically achieves a sampling rate between three and twenty times faster than the N-ary FRP implementation.

8.3 Change Properties

This section considers properties of signals and signal functions that could be exploited by an implementation to enable the kinds of optimisations suggested in Section 8.1. These are time-varying properties; that is, they may hold at some points in time and not others. They are expressed in this way to cater for optimisation opportunities that arise dynamically, rather than just those that are valid statically.

Many of the properties defined in this section are, in isolation, fairly intuitive. However, the interactions between properties, and recognising which properties hold for specific signal functions, can be quite subtle. In particular, the combination of structural dynamism with continuous and discrete time can lead to quite counter-intuitive properties. Consequently, formally expressing and reasoning about these properties gives a much sounder basis for optimisation than relying on an intuitive understanding of change.

8.3.1 Unchanging Signals

Many of the proposed optimisations rely on some notion of signals *changing*. However, most obvious definitions of change are implementation specific. For example, in a sampled implementation an obvious definition would be to say that a signal has changed if its current sample differs from its previous sample. Yet in the conceptual model of N-ary FRP there is no sequence of time samples. Also, while this would make sense for Continuous or Step signals, it would be dubious for Event signals. Two identical event occurrences that happen to be adjacent given some specific sampling strategy should be considered two changes, not a lack of change. Consequently, a more precise definition of change is needed: one that respects the conceptual model and its multi-kinded signals.

Unchanging at a Point

A signal being *unchanging* at a time point can be defined at the conceptual level as follows:

- A Continuous signal is *unchanging* at a time t if there exists a previous time point t_0 such that the signal is constant over the interval $[t_0, t]$.
- An Event signal is *unchanging* at all time points at which there is no event occurrence.
- A Step signal is *unchanging* at all time points except $time_0$ and the points at which it assumes a new value.
- A signal vector is *unchanging* if all signals in that vector are *unchanging*.

These definitions can be formulated as follows (the utility functions used can be found in appendices A and B):

```
UnchangingCP : ChangePrefix A → TPred
UnchangingCP cp t = IsNothing (lookupCP cp t)

UnchangingE : SigVec (E A) → TPred
UnchangingE (ma, cp) t | t ≡ 0 = IsNothing ma
                       | t > 0 = UnchangingCP cp t

UnchangingS : SigVec (S A) → TPred
UnchangingS (_, cp) t | t ≡ 0 = False
                      | t > 0 = UnchangingCP cp t

UnchangingC : SigVec (C A) → TPred
UnchangingC s t = P (λ t0 → ConstantOver s [t0, t]) t

Unchanging : {as : SVDesc} → SigVec as → TPred
Unchanging {C _} s          = UnchangingC s
Unchanging {S _} s          = UnchangingS s
Unchanging {E _} s          = UnchangingE s
Unchanging {_, _} (s1, s2)  = Unchanging s1 ∧ Unchanging s2
```

Note that whether a Continuous signal is *unchanging* only depends on the signal *up to and including* the time point. This is done deliberately to keep the definition causal: whether a signal is *unchanging* at a time point should not depend on future values of that signal.

If a signal is not *unchanging* at a time point, then it is said to be *changing* at that time point. Observe that both Continuous and Step signals are defined to be *changing* at $time_0$; this choice is discussed in Section 8.5.3.

Unchanging over an Interval

Next, a signal being unchanging over an interval is considered:

```
UnchangingOver : {as : SVDesc} → SigVec as → (t1 t2 : Time) → Set
UnchangingOver {C _} s          t1 t2 = ConstantOver s [t1, t2]
UnchangingOver {E _} (_, cp)    t1 t2 = t1 < t2 → cp t1 ≡ cp t2
UnchangingOver {S _} (_, cp)    t1 t2 = t1 < t2 → cp t1 ≡ cp t2
UnchangingOver {_, _} (s1, s2)  t1 t2 = UnchangingOver s1 t1 t2 × UnchangingOver s2 t1 t2
```

Intuitively, *UnchangingOver* s t_1 t_2 holds if there are no changes in signal vector s over the interval $(t_1, t_2]$.

A left-open interval is chosen as that is simpler to express than a closed interval. A reflexive variant of this property (corresponding to a closed interval) can then be defined in terms of *UnchangingOver* and *Unchanging*:

$$\begin{array}{l} \mathit{UnchangingOver}^r \;:\; \mathit{SigVec}\ \mathit{as} \to (t_1\ t_2\ :\ \mathit{Time}) \to \mathit{Set} \\ \mathit{UnchangingOver}^r\ s\ t_1\ t_2 \;=\; t_1 \leqslant t_2 \to \mathit{Unchanging}\ s\ t_1 \times \mathit{UnchangingOver}\ s\ t_1\ t_2 \end{array}$$

Changeless Signals

The preceding notions of signal change can be used to express the property of a signal vector being unchanging henceforth. Both a reflexive (including the current time) and a non-reflexive (excluding the current time) variant are defined.

A signal vector is *changeless* at a point in time if it is *unchanging between* that point and all future time points:

$$\begin{array}{l} \mathit{ChangelessSV} \;:\; \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\ \mathit{ChangelessSV}\ s\ t \;=\; \mathbf{G}\ (\mathit{UnchangingOver}\ s\ t)\ t \end{array}$$

A signal vector is *reflexively changeless* if it is both *unchanging* and *changeless*:

$$\begin{array}{l} \mathit{ChangelessSV}^r \;:\; \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\ \mathit{ChangelessSV}^r\ s \;=\; \mathit{Unchanging}\ s \wedge \mathit{ChangelessSV}\ s \end{array}$$

Aside

It may seem that the *unchanging over* property is superfluous, and that the *changeless* property could be defined by extending the *unchanging* property using the **G** combinator:

$$\begin{array}{l} \mathit{ChangelessSV}' \;:\; \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\ \mathit{ChangelessSV}'\ s \;=\; \mathbf{G}\ (\mathit{Unchanging}\ s) \end{array}$$

This would be valid for Step and Event signals, but not for Continuous signals. The problem stems from continuous time being *dense*. This is best seen by counter example.

Consider the following continuous-time signal:

$$\begin{array}{l} \mathit{gt5} \;:\; \mathit{SigVec}\ (\mathsf{C}\ \mathit{Bool}) \\ \mathit{gt5}\ t \;=\; t > 5 \end{array}$$

This signal clearly changes in value. Yet it changes *immediately after* a time point, not *at* a time point. Consequently, based on the definition of a Continuous signal being *unchanging*, there is no point in time at which this signal is *changing*. As the temporal operator **G** extends a temporal predicate over all future *points* in time, this means that the change would not be detected. To give a concrete example, *ChangelessSV′ gt5 3* holds, when, intuitively, it should not.

8.3.2 Another Pointwise Signal Equality

Section 6.5.1 introduced a notion of time-varying signal equality based on pointwise equality of samples. While suitable for expressing properties such as statelessness, this equality is not strong enough when reasoning about change. The problem is that the value of a Step signal may change to the same value it held previously; that is, the change list may contain two equal values consecutively. By sample equality, such a signal would be considered equal to an (otherwise identical) signal that lacked the second change. This is unsatisfactory, as when reasoning about change it is important to distinguish between the presence and absence of change.

One could argue that the fault is in the definition of *unchanging*, and that a Step signal should be considered to be *unchanging* in situations where the value of the change is the same as the preceding value. However, this would not be as practical for optimisation. With such a definition, knowing whether a Step signal is *unchanging* relies on being able to compare two values for equality. As Step signals are polymorphic in the type of their value, this would require comparing values of any type for equality, which is in general undecidable.

Consequently, a stronger notion of pointwise signal-vector equality is required. First, the notion of a *representation* of a signal vector at a specific point in time is defined, along with a function to compute that representation:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{SVRep \;:\; SVDesc \rightarrow Set} \\
SVRep\;(\mathsf{C}\;A) & = & A \\
SVRep\;(\mathsf{E}\;A) & = & Maybe\;A \times ChangeList\;A \\
SVRep\;(\mathsf{S}\;A) & = & A \times ChangeList\;A \\
SVRep\;(as, bs) & = & SVRep\;as \times SVRep\;bs
\end{array}
$$

$$
\begin{array}{llcl}
\multicolumn{4}{l}{rep \;:\; \{as \;:\; SVDesc\} \rightarrow SigVec\;as \rightarrow Time \rightarrow SVRep\;as} \\
rep\;\{\mathsf{C}\;\_\}\;s & t & = & s\;t \\
rep\;\{\mathsf{E}\;\_\}\;(ma, cp) & t & = & (ma, cp\;t) \\
rep\;\{\mathsf{S}\;\_\}\;(a, cp) & t & = & (a, cp\;t) \\
rep\;\{\_,\_\}\;(s_1, s_2) & t & = & (rep\;s_1\;t, rep\;s_2\;t)
\end{array}
$$

Intuitively, this is just applying all time-varying elements of the signal vector to a specific time value.

A pointwise equality of signal vector representations is then defined:

$$
\begin{array}{l}
EqRep \;:\; SigVec\;as \rightarrow SigVec\;as \rightarrow TPred \\
EqRep\;s_1\;s_2 \;=\; rep\;s_1 \;\dot{=}\; rep\;s_2
\end{array}
$$

As mentioned, this is a stronger property than sample equality:

$$EqRep\;s_1\;s_2 \Rightarrow EqSample\;s_1\;s_2$$

Pragmatically, it is also much easier to reason with, principally due to the absence of the auxiliary functions *val* and *occ*.

Finally, many of the properties in this section depend on a variant formulation of causality that takes into account representation equality:

$$
\begin{array}{l}
RepCausal \;:\; SF\;as\;bs \rightarrow Set \\
RepCausal\;sf \;=\; \forall\;s_1\;s_2 \rightarrow Always\;(\mathbf{H^r}\;(EqRep\;s_1\;s_2) \Rightarrow EqRep\;(sf\;s_1)\;(sf\;s_2))
\end{array}
$$

This property holds for all primitive signal functions in N-ary FRP, and is preserved by all primitive combinators in the same way as causality based on sample equality.

8.3.3 Change Properties of Signal Functions

This section defines several properties of signal functions. Unlike the properties in Section 6.5, these are dynamic properties that may come to hold during execution. Thus whether they hold for a given signal function varies with time, and also depends on the input the signal function has received up to that time point. Consequently, these properties are expressed as temporal predicates parametrised over both a signal function and its input signal vector.

Changeless Signal Functions

Signal functions that produce *changeless* and *reflexively changeless* signal vectors are expressed as follows:

$$
\begin{array}{l}
\mathit{Changeless} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\
\mathit{Changeless}\ \mathit{sf}\ s\ t\ =\ \forall\, s' \to (\mathbf{H^r}\ (\mathit{EqRep}\ s\ s') \Rightarrow \mathit{ChangelessSV}\ (\mathit{sf}\ s'))\ t \\
\mathit{Changeless}^r : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\
\mathit{Changeless}^r\ \mathit{sf}\ s\ t\ =\ \forall\, s' \to (\mathbf{H}\ (\mathit{EqRep}\ s\ s') \Rightarrow \mathit{ChangelessSV}^r\ (\mathit{sf}\ s'))\ t
\end{array}
$$

These definitions can be read as saying that a signal function *sf*, having been applied to a signal vector s, is *changeless* at a point in time t, if, for any signal vector s' such that s' and s have been identical up to time t, $\mathit{sf}\ s'$ will be unchanging henceforth. Or, more intuitively, no matter what input is received in the future, the output will be unchanging henceforth.

Change-Propagating Signal Functions

More interesting are signal functions that *propagate* change; that is, signal functions that produce unchanging output when given unchanging input. This idea can be split into two main properties, called *change-dependent* and *change-propagating*. A *change-propagating* signal function will produce unchanging output over any period in the future for which its input is unchanging. A *change-dependent* signal function will do likewise, but only over periods that start at the current time point. Thus *change-dependent* is a strictly weaker property. These properties can be formulated as follows (with reflexive variants as usual):

$$
\begin{array}{l}
\mathit{ChangeDep} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\
\mathit{ChangeDep}\ \mathit{sf}\ s\ t\ =\ \forall\, s' \to \\
\quad (\mathbf{H^r}\ (\mathit{EqRep}\ s\ s') \Rightarrow \mathbf{G}\ (\mathit{UnchangingOver}\ s'\ t \Rightarrow \mathit{UnchangingOver}\ (\mathit{sf}\ s')\ t))\ t \\
\\
\mathit{ChangeDep}^r : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\
\mathit{ChangeDep}^r\ \mathit{sf}\ s\ t\ =\ \forall\, s' \to \\
\quad (\mathbf{H}\ (\mathit{EqRep}\ s\ s') \Rightarrow \mathbf{G^r}\ (\mathit{UnchangingOver}^r\ s'\ t \Rightarrow \mathit{UnchangingOver}^r\ (\mathit{sf}\ s')\ t))\ t \\
\\
\mathit{ChangePrp} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\
\mathit{ChangePrp}\ \mathit{sf}\ s\ =\ \mathbf{G^r}\ (\mathit{ChangeDep}\ \mathit{sf}\ s) \wedge \mathbf{G}\ (\mathit{ChangeDep}^r\ \mathit{sf}\ s) \\
\\
\mathit{ChangePrp}^r : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\
\mathit{ChangePrp}^r\ \mathit{sf}\ s\ =\ \mathit{ChangePrp}\ \mathit{sf}\ s \wedge \mathit{ChangeDep}^r\ \mathit{sf}\ s
\end{array}
$$

Recall that *UnchangingOver* is concerned with *left-open* intervals, and $\mathit{UnchangingOver}^r$ with *left-closed* intervals. Thus *ChangeDep* is concerned with left-open intervals and $\mathit{ChangeDep}^r$ with left-closed intervals. *ChangePrp* and $\mathit{ChangePrp}^r$ are concerned with both (hence the conjunction), differing only in that *ChangePrp* excludes left-closed intervals starting at the current time, whereas $\mathit{ChangePrp}^r$ includes them.

Finally, note that the main reason for distinguishing between *change-dependent* and *change-propagating* is that greater precision is then possible when expressing the properties of *switch* (see Section 8.3.5).

Source Signal Functions

Signal functions that no longer use their input are called *sources*. More precisely, a signal function is a *source* at a time t if its output after time t does not depend on input after time

t. As ever, this is split into reflexive and non-reflexive variants that include and exclude the current time:

$$
\begin{aligned}
&\mathit{Source} : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred}\\
&\mathit{Source}\ \mathit{sf}\ s\ t = \forall\, s' \to (\mathbf{H^r}\ (\mathit{EqRep}\ s\ s') \Rightarrow \mathbf{G}\ (\mathit{EqRep}\ (\mathit{sf}\ s)\ (\mathit{sf}\ s')))\ t\\
&\mathit{Source}^r : \mathit{SF}\ \mathit{as}\ \mathit{bs} \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred}\\
&\mathit{Source}^r\ \mathit{sf}\ s\ t = \forall\, s' \to (\mathbf{H}\ (\mathit{EqRep}\ s\ s') \Rightarrow \mathbf{G^r}\ (\mathit{EqRep}\ (\mathit{sf}\ s)\ (\mathit{sf}\ s')))\ t
\end{aligned}
$$

8.3.4 Implications between Properties

Many of the change properties in the preceding section are strictly stronger or weaker than each other. The properties that imply other properties are listed below:

$$
\begin{aligned}
&\mathit{Changeless}^r\ \mathit{sf}\ s &&\Rightarrow \mathit{Changeless}\ \mathit{sf}\ s\\
&\mathit{Changeless}^r\ \mathit{sf}\ s &&\Rightarrow \mathit{ChangePrp}^r\ \mathit{sf}\ s\\
&\mathit{Changeless}^r\ \mathit{sf}\ s &&\Rightarrow \mathit{Source}^r\ \mathit{sf}\ s\\
&\mathit{Changeless}\ \mathit{sf}\ s &&\Rightarrow \mathit{ChangePrp}\ \mathit{sf}\ s\\
&\mathit{Changeless}\ \mathit{sf}\ s &&\Rightarrow \mathit{Source}\ \mathit{sf}\ s\\
&\mathit{ChangePrp}^r\ \mathit{sf}\ s &&\Rightarrow \mathit{ChangeDep}^r\ \mathit{sf}\ s\\
&\mathit{ChangePrp}^r\ \mathit{sf}\ s &&\Rightarrow \mathit{ChangePrp}\ \mathit{sf}\ s\\
&\mathit{ChangePrp}\ \mathit{sf}\ s &&\Rightarrow \mathit{ChangeDep}\ \mathit{sf}\ s\\
&\mathit{Source}^r\ \mathit{sf}\ s &&\Rightarrow \mathit{Source}\ \mathit{sf}\ s
\end{aligned}
$$

Note that *reflexively change-dependent* does not imply *change-dependent*. Intuitively, this is because there could be a change in the input at the current time point.

For most change properties, if they hold at a time point then they continue to hold thereafter. The exceptions are the *change-dependent* and *reflexively change-dependent* properties. This can be summarised as follows:

$$
\begin{aligned}
&\mathit{Changeless}\ \mathit{sf}\ s &&\Rightarrow \mathbf{G}\ (\mathit{Changeless}\ \mathit{sf}\ s)\\
&\mathit{Changeless}^r\ \mathit{sf}\ s &&\Rightarrow \mathbf{G}\ (\mathit{Changeless}^r\ \mathit{sf}\ s)\\
&\mathit{ChangePrp}\ \mathit{sf}\ s &&\Rightarrow \mathbf{G}\ (\mathit{ChangePrp}\ \mathit{sf}\ s)\\
&\mathit{ChangePrp}^r\ \mathit{sf}\ s &&\Rightarrow \mathbf{G}\ (\mathit{ChangePrp}^r\ \mathit{sf}\ s)\\
&\mathit{Source}\ \mathit{sf}\ s &&\Rightarrow \mathbf{G}\ (\mathit{Source}\ \mathit{sf}\ s)\\
&\mathit{Source}^r\ \mathit{sf}\ s &&\Rightarrow \mathbf{G}\ (\mathit{Source}^r\ \mathit{sf}\ s)
\end{aligned}
$$

Finally, if a signal function is both *change-dependent* and a *source*, then it is *changeless*:

$$\mathit{ChangeDep}\ \mathit{sf} \land \mathit{Source}\ \mathit{sf} \Rightarrow \mathit{ChangelessSF}\ \mathit{sf}$$

Aside

Somewhat counter-intuitively, the reflexive variant of the preceding implication does not hold. A counter example is the following specialisation of the *now* primitive:

$$
\begin{aligned}
&\mathit{nowS} : \mathit{SF}\ (\mathsf{S}\ \mathit{Unit})\ (\mathsf{E}\ \mathit{Unit})\\
&\mathit{nowS} = \mathit{const}\ (\mathsf{just\ unit}, \mathit{const}\ [\,])
\end{aligned}
$$

This signal function is both *reflexively change-dependent* (it only produces changing output at $time_0$, when the input is *changing*), and a *reflexive source* (it never uses its input) at all points in time. Yet at $time_0$ it is not *reflexively changeless*, as it emits an event (a change). The issue is that there is no possible input signal vector that is unchanging at $time_0$, as Step signals are always *changing* at $time_0$. Consequently, the *change-dependent* property holds when, intuitively, it should not.

8.3.5 Properties of N-ary FRP Primitives

This section considers which of the properties in the preceding section hold for the N-ary FRP primitives.

Primitive Signal Functions

The properties that hold for the primitive signal functions at all points in time and for all input signal vectors are as follows:

- $\mathit{Changeless}$: $\mathit{constantS}$, never, now
- $\mathit{Changeless}^r$: never
- $\mathit{ChangeDep}$: $\mathit{constantS}$, never, now, notYet, $\mathit{filterE}$, hold, edge, fromS
- $\mathit{ChangeDep}^r$: never, notYet, $\mathit{filterE}$, edge, fromS
- $\mathit{ChangePrp}$: $\mathit{constantS}$, never, now, notYet, $\mathit{filterE}$, hold, edge, fromS
- $\mathit{ChangePrp}^r$: never, notYet, $\mathit{filterE}$, edge, fromS
- Source: $\mathit{constantS}$, never, now
- Source^r: $\mathit{constantS}$, never, now

Lifting Functions and Atomic Routers

At all points in time and for all input signal vectors, the following properties hold for the atomic routers and for all signal functions produced by the lifting functions:

$$\mathit{ChangeDep}, \mathit{ChangeDep}^r, \mathit{ChangePrp}, \mathit{ChangePrp}^r$$

Routing Combinators

All of the properties are preserved by the routing combinators:

$$
\begin{array}{lll}
\mathit{Changeless}\ sf_1\ s & \wedge\ \mathit{Changeless}\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Changeless}\ (sf_1 \ggg sf_2)\ s \\
\mathit{Changeless}^r\ sf_1\ s & \wedge\ \mathit{Changeless}^r\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Changeless}^r\ (sf_1 \ggg sf_2)\ s \\
\mathit{ChangeDep}\ sf_1\ s & \wedge\ \mathit{ChangeDep}\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{ChangeDep}\ (sf_1 \ggg sf_2)\ s \\
\mathit{ChangeDep}^r\ sf_1\ s & \wedge\ \mathit{ChangeDep}^r\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{ChangeDep}^r\ (sf_1 \ggg sf_2)\ s \\
\mathit{ChangePrp}\ sf_1\ s & \wedge\ \mathit{ChangePrp}\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{ChangePrp}\ (sf_1 \ggg sf_2)\ s \\
\mathit{ChangePrp}^r\ sf_1\ s & \wedge\ \mathit{ChangePrp}^r\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{ChangePrp}^r\ (sf_1 \ggg sf_2)\ s \\
\mathit{Source}\ sf_1\ s & \wedge\ \mathit{Source}\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Source}\ (sf_1 \ggg sf_2)\ s \\
\mathit{Source}^r\ sf_1\ s & \wedge\ \mathit{Source}^r\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Source}^r\ (sf_1 \ggg sf_2)\ s
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Changeless}\ sf_1\ s & \wedge\ \mathit{Changeless}\ sf_2\ s & \Rightarrow \mathit{Changeless}\ (sf_1 \mathbin{\&\&\&} sf_2)\ s \\
\mathit{Changeless}^r\ sf_1\ s & \wedge\ \mathit{Changeless}^r\ sf_2\ s & \Rightarrow \mathit{Changeless}^r\ (sf_1 \mathbin{\&\&\&} sf_2)\ s \\
\mathit{ChangeDep}\ sf_1\ s & \wedge\ \mathit{ChangeDep}\ sf_2\ s & \Rightarrow \mathit{ChangeDep}\ (sf_1 \mathbin{\&\&\&} sf_2)\ s \\
\mathit{ChangeDep}^r\ sf_1\ s & \wedge\ \mathit{ChangeDep}^r\ sf_2\ s & \Rightarrow \mathit{ChangeDep}^r\ (sf_1 \mathbin{\&\&\&} sf_2)\ s \\
\mathit{ChangePrp}\ sf_1\ s & \wedge\ \mathit{ChangePrp}\ sf_2\ s & \Rightarrow \mathit{ChangePrp}\ (sf_1 \mathbin{\&\&\&} sf_2)\ s \\
\mathit{ChangePrp}^r\ sf_1\ s & \wedge\ \mathit{ChangePrp}^r\ sf_2\ s & \Rightarrow \mathit{ChangePrp}^r\ (sf_1 \mathbin{\&\&\&} sf_2)\ s \\
\mathit{Source}\ sf_1\ s & \wedge\ \mathit{Source}\ sf_2\ s & \Rightarrow \mathit{Source}\ (sf_1 \mathbin{\&\&\&} sf_2)\ s \\
\mathit{Source}^r\ sf_1\ s & \wedge\ \mathit{Source}^r\ sf_2\ s & \Rightarrow \mathit{Source}^r\ (sf_1 \mathbin{\&\&\&} sf_2)\ s
\end{array}
$$

In the case of sequential composition, there are also some stronger properties:

$$
\begin{array}{lll}
 & \mathit{Changeless}\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Changeless}\ (sf_1 \ggg sf_2)\ s \\
 & \mathit{Changeless}^r\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Changeless}^r\ (sf_1 \ggg sf_2)\ s \\
\mathit{Changeless}\ sf_1\ s & \wedge\ \mathit{ChangeDep}\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Changeless}\ (sf_1 \ggg sf_2)\ s \\
\mathit{Changeless}^r\ sf_1\ s & \wedge\ \mathit{ChangeDep}^r\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Changeless}^r\ (sf_1 \ggg sf_2)\ s \\
\mathit{Source}\ sf_1\ s & & \Rightarrow \mathit{Source}\ (sf_1 \ggg sf_2)\ s \\
\mathit{Source}^r\ sf_1\ s & & \Rightarrow \mathit{Source}^r\ (sf_1 \ggg sf_2)\ s \\
 & \mathit{Source}\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Source}\ (sf_1 \ggg sf_2)\ s \\
 & \mathit{Source}^r\ sf_2\ (sf_1\ s) & \Rightarrow \mathit{Source}^r\ (sf_1 \ggg sf_2)\ s
\end{array}
$$

Freeze

The *freeze* combinator preserves the following properties:

$$\begin{array}{lll} \mathit{Changeless}^r\ \mathit{sf}\ s & \Rightarrow & \mathit{Changeless}^r\ (\mathit{freeze}\ \mathit{sf})\ s \\ \mathit{Source}\ \mathit{sf}\ s & \Rightarrow & \mathit{Source}\ (\mathit{freeze}\ \mathit{sf})\ s \\ \mathit{Source}^r\ \mathit{sf}\ s & \Rightarrow & \mathit{Source}^r\ (\mathit{freeze}\ \mathit{sf})\ s \end{array}$$

As with *statelessness* (Section 6.5.4), these properties of *freeze* rely on signal function extensionality. The frozen signal function at any time point is never intensionally equal to the same signal function frozen at any other time point, and hence the Continuous signal that carries it is always *changing* under intensional equality.

Finally, some properties are preserved (or weakened) in a frozen signal function, and then hold at the moment it is switched-in again (local $time_0$):

$$\begin{array}{lll} \mathit{Changeless}^r\ \mathit{sf}\ s & \Rightarrow & (\lambda\ t \to (\forall\ s' \to \mathit{Changeless}\ (\mathit{frozenSample}\ \mathit{sf}\ s\ t)\ s'\ 0)) \\ \mathit{ChangePrp}^r\ \mathit{sf}\ s & \Rightarrow & (\lambda\ t \to (\forall\ s' \to \mathit{ChangePrp}\ (\mathit{frozenSample}\ \mathit{sf}\ s\ t)\ s'\ 0)) \\ \mathit{Source}^r\ \mathit{sf}\ s & \Rightarrow & (\lambda\ t \to (\forall\ s' \to \mathit{Source}^r\ (\mathit{frozenSample}\ \mathit{sf}\ s\ t)\ s'\ 0)) \end{array}$$

Note that $\mathit{Changeless}^r$ and $\mathit{ChangePrp}^r$ are weakened to *Changeless* and *ChangePrp*. This is because a Step or Continuous signal that is *unchanging* at the point it is switched-out is nevertheless considered *changing* at the local $time_0$ when it is switched-in.

The non-reflexive properties do not imply anything about the frozen signal function, as they are properties that hold at a time t if the signal function has been applied to the input signal vector up to and *including* time t. A frozen signal function has been applied to input up to yet *excluding* the time point at which it was frozen, and thus the non-reflexive properties are not preserved in the frozen signal function.

Switch

Because of its dynamic nature, preservation of properties by the *switch* combinator is slightly more involved. Essentially, the issue is that the time-varying properties of *switch* depend on whether or not the switch has occurred yet. To express this, some auxiliary predicates are required:

$$\begin{array}{l} \mathit{NoOccs} : \mathit{SigVec}\ (\mathsf{E}\ A) \to \mathit{TPred} \\ \mathit{NoOccs}\ s\ t = \mathit{fstOcc}\ s\ t \equiv \mathsf{nothing} \\[4pt] \mathit{FstOcc} : \mathit{Time} \times A \to \mathit{SigVec}\ (\mathsf{E}\ A) \to \mathit{TPred} \\ \mathit{FstOcc}\ e\ s\ t = \mathit{fstOcc}\ s\ t \equiv \mathsf{just}\ e \\[4pt] \mathit{NotSwitched} : \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A) \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\ \mathit{NotSwitched}\ \mathit{sf}\ s = \mathit{NoOccs}\ (\mathit{snd}\ (\mathit{sf}\ s)) \\[4pt] \mathit{Switched} : \mathit{Time} \times A \to \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A) \to \mathit{SigVec}\ \mathit{as} \to \mathit{TPred} \\ \mathit{Switched}\ e\ \mathit{sf}\ s = \mathit{FstOcc}\ e\ (\mathit{snd}\ (\mathit{sf}\ s)) \end{array}$$

These can be understood as follows:

- $\mathit{NoOccs}\ s\ t$ expresses that no events have occurred within Event signal s up to time t.
- $\mathit{FstOcc}\ (t_e, a)\ s\ t$ expresses that at least one event has occurred within Event signal s up to time t, and that the first such event occurred at time t_e with value a.
- $\mathit{NotSwitched}\ \mathit{sf}\ s\ t$ expresses that signal function sf, having been applied to signal vector s, has not produced any event occurrences up to time t in its Event signal output.

- *Switched* (t_e, a) *sf s t* expresses that the signal function *sf*, having been applied to signal vector s, has produced at least one event occurrence up to time t in its Event signal output, and that the first such event occurred at time t_e with value a.

Note that *FstOcc* and *Switched* are parametrised over the time and value of the first event occurrence in order to simplify expressing the forthcoming properties of *switch*.

Considering first the case where the switch has not yet occurred, the properties of *switch* are as follows:

$$\begin{array}{lcl}
\mathit{NotSwitched}\ \mathit{sf}\ s \wedge \mathit{Changeless}\ \mathit{sf}\ s & \Rightarrow & \mathit{Changeless}\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{NotSwitched}\ \mathit{sf}\ s \wedge \mathit{Changeless}^r\ \mathit{sf}\ s & \Rightarrow & \mathit{Changeless}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{NotSwitched}\ \mathit{sf}\ s \wedge \mathit{ChangeDep}\ \mathit{sf}\ s & \Rightarrow & \mathit{ChangeDep}\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{NotSwitched}\ \mathit{sf}\ s \wedge \mathit{ChangeDep}^r\ \mathit{sf}\ s & \Rightarrow & \mathit{ChangeDep}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{NotSwitched}\ \mathit{sf}\ s \wedge \mathit{ChangePrp}\ \mathit{sf}\ s & \Rightarrow & \mathit{ChangeDep}\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{NotSwitched}\ \mathit{sf}\ s \wedge \mathit{ChangePrp}^r\ \mathit{sf}\ s & \Rightarrow & \mathit{ChangeDep}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s
\end{array}$$

In general, the change properties of the residual signal function are not known until the switch occurs; thus implications that rely on knowing the properties of the residual signal function have been omitted. Also, note that the *change-propagating* properties are not fully preserved, but weakened to *change-dependent*. This is because the *change-propagating* properties can be lost when a switch occurs, but a switch cannot occur until there is a change in the input.

Finally, if the switch has occurred, then the properties of the switching combinator will be those of the residual signal function. Intuitively this is fairly simple, but expressing it formally is slightly awkward as the signals have to be translated into the local time frame of the residual signal function:

$$\begin{array}{lcl}
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{Changeless}\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{Changeless}\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{ChangeDep}\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{ChangeDep}\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{ChangePrp}\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{ChangePrp}\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{Source}\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{Source}\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{Changeless}^r\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{Changeless}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{ChangeDep}^r\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{ChangeDep}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{ChangePrp}^r\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{ChangePrp}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{Source}^r\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow & \mathit{Source}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s
\end{array}$$

8.4 Implementing Signal Function Properties

As discussed in Section 3.5.4, one of the advantages of a first-class signal-function abstraction is that additional information can be associated with it. Thus signal functions can record internally which properties they satisfy. Provided the implementer identifies the properties of all the primitives, the properties of any composite signal function can be computed from those of its components, using the implications in Section 8.3.5.

In most cases these properties would be kept internal to the implementation, and used only for optimisation purposes. That said, there are cases when it can be advantageous to make some properties visible. For example, as discussed in Chapter 7, the *decoupled* property can be encoded in the type system, thereby allowing the type-checker to ensure the absence of instantaneous feedback within the network. In the context of FRP, the main advantage of encoding properties in the type system is that properties of residual signal functions can be inferred before the values of those signal functions are computed.

8.5 Suggested Optimisations

This section overviews the change-based optimisations that are possible on a signal function network. How precisely to *implement* such optimisations is not discussed, as that depends on the details of the specific FRP implementation involved. Some optimisations may be more applicable for some implementations than others.

8.5.1 Structural Optimisation

As discussed in Section 8.1.2, structural optimisation involves eliminating unnecessary signal functions and combinators from the network. There are two issues to consider: *when* structural optimisations could be applied, and *what* structural optimisations are possible. I reiterate that the scope of this chapter is only change-based structural optimisations that follow from the properties in Section 8.3; there are, of course, many other structural-optimisation techniques that can be applied to signal function networks (such as *lowering* [14, 23] or *causal-commutative-arrow normalisation* [75, 77]).

When to Optimise?

Structural optimisations can be applied either statically (at compile time), or dynamically (at run-time). Static optimisations are appealing because they incur no run-time cost. However, because of FRP's dynamic nature, many optimisations can only be applied at run-time. Furthermore, dynamic optimisations have a lot more potential for simplifying the network. For signal functions to be eliminated statically, the programmer must have included unnecessary code in her program. But, as previously mentioned, it is common for signals to cease to be used, or to become constant, as a result of structural switches during execution. Thus, even programs that contain no unnecessary code will produce "dead" signal functions during execution. Such signal functions can be eliminated dynamically, often significantly reducing the network size.

For dynamic optimisations, there is still a choice as to when, precisely, to apply them. One could imagine either applying them at a small number of specific time points, or continually attempting to optimise during execution.

For example, the most recent version of Yampa [90] takes the latter approach by attempting structural optimisation at every time step. However, it does not try to *fully* optimise at each step. Essentially, if an opportunity for an optimisation is detected then it is applied, but further optimisation opportunities that this may have created are not checked for. Instead, they will be detected, and applied, at the next time step. This leads to a situation where one step of optimisation is applied at each time step, until no further optimisation is possible. However, even when the network has been fully optimised, optimisation opportunities are still being checked for. Thus, whenever a structural switch occurs, if any further optimisations become possible, they begin to be applied. The advantage of this approach is that the cost of optimisation is spread out over several time steps, rather than producing a time-lag at a particular point in time. The disadvantage is that computation is wasted by continually checking for optimisations even after the network has been fully optimised.

The alternative approach is to optimise only at certain time points: when the network is first initialised and at each structural switch thereafter. Between these points in time the network remains static, and thus new optimisation opportunities will not arise[1]. Note that when structural optimisations are applied dynamically, the network is restructured for all future points in time, *excluding* the present. Or, to put it more operationally, optimisation is performed *after* execution of the initialisation step. In the case of newly switched-in signal functions, one could imagine optimising their structure including the present time. However, there are more optimisation opportunities if just the future is optimised. In essence, the reflexive variants of the change properties determine which optimisations are valid *including* the current time, and the non-reflexive variants determine which optimisations are valid *excluding* the current time. More primitives have the non-reflexive properties, and thus more optimisations are valid if the current time is excluded.

If a network is only optimised when a structural switch occurs, then there is a further choice as to what parts of the network to optimise at those points. Should the entire network be optimised, or just the newly switched-in signal function? If the network is large, the former option could be computationally expensive. However, not doing so will miss some optimisation opportunities. Note that this question does not arise if optimisations are applied continually, as the entire network is continually being optimised. One solution to this dilemma is to optimise the network in the locality of the switch, iterating outwards until no more optimisations are possible. An algorithm for this can be summarised as follows:

1. Optimise the residual signal function.
2. Compare the change properties of the residual signal function with those of the switching combinator it has replaced. If it has the same or weaker properties, then stop. Otherwise, optimise the sub-network containing the residual signal function.
3. Compare the change properties of this sub-network with those it had prior to optimisation. If it has the same or weaker properties, then stop. Otherwise, optimise the sub-network containing this sub-network, and repeat.

Constant Propagation

If a Step or Continuous signal is *changeless* at a point in time, then it is constant thereafter. Repeatedly recomputing a constant value is a waste of computational resources and should be avoided. Similarly, a *changeless* Event signal contains no future event occurrences. A *changeless* signal function produces *changeless* signals, and thus can be eliminated from the network (and garbage collected). Constant propagation can then be applied, propagating the value of the output sample at the current point in time (in the case of Step or Continuous signals), or the absence of event occurrences (in the case of Event signals), throughout the network. This is a dynamic optimisation.

[1]At least, not for the optimisations considered here. One could imagine more fine-grained optimisations specialised to individual primitives.

There is a similar static optimisation for *reflexively changeless* signal functions. If a signal function is *reflexively changeless* at $time_0$, then there cannot be an occurrence in any output Event signals at $time_0$ (to be *reflexively changeless* at $time_0$, the outputs cannot be Step or Continuous signals). Thus the signal function can be eliminated, and the empty Event signals propagated.

In both cases, the constant propagation can lead to other signal functions becoming sources if all of their (used) inputs are set as constant. This can then present further optimisation opportunities. In the dynamic case they gain the *source* property, in the static case they gain the *reflexive source* property.

A limited form of dynamic constant propagation is employed by the most recent version of Yampa [90].

Eliminating Unused Signal Functions

Any signal function whose output is not used can be eliminated. This could arise either because the signals are eliminated by routing primitives (and thus never reach another signal function), or because all signal functions that do receive it are sources. This is essentially reactive-level garbage collection, exploiting the properties of routing combinators and signal functions to identify unused signal functions.

This can be applied as a static optimisation, in which case signal functions can be determined not to use their input if they are *reflexive sources* at $time_0$. It can also be applied as a dynamic optimisation, in which case signal functions are determined not to use their input if they are *sources* at the time of optimisation.

The most recent version of Yampa uses this technique to some degree [90], but is limited by the UFRP model. As discussed in Section 3.5.2, much of the routing of the arrow framework is carried out by lifted pure functions, hiding the routing from the reactive level.

Switch Elimination

A *switch* combinator, and its subordinate signal function, can be eliminated after the structural switch occurs, leaving just the residual signal function in its place. This dynamic optimisation is so natural and simple that it is often easier to implement the *switch* combinator with this optimisation than without (as it avoids the need to remember if the structural switch has occurred). The Haskell embedding of N-ary FRP does this (see Appendix C.2).

More interestingly, switching combinators for which a structural switch will never occur can also be eliminated. Statically, if the Event signal produced by the subordinate signal function of *switch* is *reflexively changeless* at $time_0$, then the subordinate signal function will never be switched-out. Dynamically, if the Event signal is *changeless*, and the switch has not yet occurred, then the subordinate signal function will never be switched-out. Consequently, in these situations, the *switch* combinator can be eliminated and replaced with the subordinate signal function (and routing primitives to discard the Event signal) as follows:

$$(\mathit{switch}\ \mathit{sf}\ f) \rightsquigarrow (\mathit{sf} \ggg \mathit{sfFst})$$

Eliminating switching combinators can be of considerable benefit, as they often obstruct other optimisation techniques (such as causal-commutative-arrow normalisation [75]).

Loop Elimination

Looping combinators that do not use the fed-back signal can be eliminated. This can be done statically if the feedback signal function is a *reflexive source* at $time_0$, or dynamically if it is a *source* at the time of optimisation. The combinator can be restructured as follows:

$(loop\ sff\ sfb)\ \rightsquigarrow\ (forkSecond\ sfb \ggg sff)$

Note that this is type incorrect, as the type of the input to *sfb* should be that of the output of *sff*. However, as a source ignores its input, its input type can be coerced to any signal vector (and any implementation of sources should reflect this).

Example of Structural Optimisation

As an example of structural optimisation, consider the following program:

$oneInelasticReset\ :\ Ball \rightarrow SF\ (\mathsf{E}\ Ball)\ (\mathsf{C}\ Ball)$
$oneInelasticReset\ b\ =\ once \ggg sfFork \ggg resetBall\ inelasticBall\ b$

This models a falling inelastic ball that may be moved to a new position at-most once in response to an external event[2]. Not counting routing primitives, this program contains ten primitive signal functions and three switching combinators. One of the switching combinators is used recursively. The two main opportunities for dynamic optimisation are when the reset event occurs, and when the ball impacts the ground thereafter. After these two occurrences, the program reduces to a constant signal defined by two primitive signal functions:

$$\begin{array}{ll}
\quad once \ggg sfFork \ggg resetBall\ inelasticBall\ b & \\
\rightsquigarrow & \{\text{switch elimination (after the reset event occurs)}\} \\
\quad nowTag\ b' \ggg sfFork \ggg resetBall\ inelasticBall\ b & \\
\rightsquigarrow & \{\text{constant propagation and switch elimination}\} \\
\quad inelasticBall\ b' & \\
\rightsquigarrow & \{\text{switch elimination (after the ball impacts the ground)}\} \\
\quad constantS\ (0,0) \ggg fromS &
\end{array}$$

8.5.2 Change Propagation

The motivation for change propagation is that many signal functions are such that their output remains unchanging while their input is unchanging. The idea is to identify where this is the case, and then not recompute the unchanging output. This approach is inherent to push-based implementations of FRP (such as Event-Driven FRP [129], FrTime [23] and Grapefruit [63]), wherein a signal is only recomputed when there is a change in a signal upon which it depends. It is also present in push-pull implementations (such as Reactive [37]) that make use of push-based execution for discrete-time signals, and pull-based implementation for continuous-time signals. However, change propagation is still possible to some degree for the continuous-time signals of such systems, and is also useful for entirely pull-based systems (such as Yampa [90]).

A common way to implement signal functions is as state transition functions in a data-flow network (this approach is taken by Yampa and the implementations in this thesis). Such transition functions execute over a discrete sequence of time steps, mapping an input sample and state to an output sample and state at each step. Each signal function maintains an

[2]The example is contrived, but not unreasonably so. A video game containing an entity that "respawns" a finite number of times would give rise to a similar situation.

internal state, rather than sharing a global state. In this pull-based style of implementation, change propagation can be applied during execution to avoid re-computation of samples, thereby regaining some of the efficiency of a push-based implementation.

Change propagation is hindered if information about which signals are *unchanging* is lost. This is a problem in Yampa where, because there is no difference between a tuple of signals and a signal of tuples (see Section 3.5.2), a change to one signal in a tuple appears to be a change to all signals in the tuple.

FrTime, on the other hand, has very effective change propagation. As well as being push-driven, it also performs run-time equality checks to compare a recomputed value with the previous value, to determine if it really has changed [24].

Change-Executable Signal Functions

A problem with applying change propagation to N-ary FRP is that the output from some signal functions can change even if its input does not (e.g. integration). Furthermore, even for *change-dependent* signal functions (which do not have that problem), there can be a change in the internal state even if the input (and thus also output) does not change. For example, consider the following *change-dependent* signal function:

$$\begin{aligned}
&\mathit{sampleTime} \;:\; \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{E}\ \mathit{Time})\\
&\mathit{sampleTime} \;=\; \mathit{forkFirst}\ \mathit{localTime} \ggg \mathit{sampleC}
\end{aligned}$$

Executing this signal function (in, say, the sampled implementation from Chapter 5) only when there is an input event occurrence would cause the output of *localTime* to "lose time"[3].

To account for this, the following stronger signal function property is required. A signal function is said to be *change-executable* if "cutting out" an unchanging interval from its input signal vector is equivalent to "cutting out" the same interval from its output signal vector:

$$\begin{aligned}
&\mathit{ChangeExec}^{r} \;:\; \mathit{SF}\ \mathit{as}\ \mathit{bs} \rightarrow \mathit{SigVec}\ \mathit{as} \rightarrow \mathit{TPred}\\
&\mathit{ChangeExec}^{r}\ \mathit{sf}\ s\ t \;=\; \forall\, s' \rightarrow \mathbf{H}\ (\mathit{EqRep}\ s\ s')\ t \rightarrow \forall\, t' \rightarrow t' \geqslant t \rightarrow \mathit{UnchangingOver}^{r}\ s'\ t\ t'\\
&\qquad\qquad\qquad \rightarrow \mathit{UnchangingOver}^{r}\ (\mathit{sf}\ s')\ t\ t' \times \mathbf{G}^{\mathbf{r}}\ (\mathit{EqRep}\ (\mathit{sf}\ (\mathit{cut}\ t\ t'\ s'))\ (\mathit{cut}\ t\ t'\ (\mathit{sf}\ s')))\ t\\
&\quad \mathbf{where}\\
&\qquad \mathit{cut} \;:\; \mathit{Time} \rightarrow \mathit{Time} \rightarrow \mathit{SigVec}\ \mathit{xs} \rightarrow \mathit{SigVec}\ \mathit{xs}\\
&\qquad \mathit{cut}\ t_1\ t_2\ \mathit{sv} \;=\; \mathit{splice}\ \mathit{sv}\ (\mathit{advance}\ t_2\ \mathit{sv})\ t_1
\end{aligned}$$

The *splice* and *advance* utility functions are defined in Appendix B.5. Intuitively, $\mathit{cut}\ t_1\ t_2\ \mathit{sv}$ "cuts-out" a segment corresponding to the interval $[t_1, t_2)$ from the signal vector sv.

One would expect to be able to define a non-reflexive variant of *change executable* as well; but, for unfortunate technical reasons that will be discussed in Section 9.1, such a property is not expressible in the N-ary FRP model. However, *change executable* is sufficient to say that, in a sampled implementation faithful to the conceptual model, it is valid to not execute a *change-executable* signal function at any time step for which its input is unchanging. The output sample at such points is the absence of an event occurrence for Event signals, and the previous output sample for Step and Continuous signals.

Formally verifying which signal functions are *change-executable* remains the subject of future work, as the use of *cut* makes reasoning about the property significantly more complex than the other change properties. That said, it is straightforward to prove that *change executable* is strictly stronger than *reflexively change-dependent*:

[3]Except in the unlikely case of an event occurring at every time step.

$$\mathit{ChangeExec}^r\ \mathit{sf}\ s \Rightarrow \mathit{ChangeDep}^r\ \mathit{sf}\ s$$

Consequently, being *change-dependent* is a necessary requirement for a signal function to be *change-executable*.

The atomic routers and the signal functions produced by the lifting primitives are *change-executable*. Also, initial investigation suggests that the following primitive signal functions are *change-executable*:

$$\mathit{never}, \mathit{notYet}, \mathit{filterE}, \mathit{edge}, \mathit{fromS}$$

Finally, it is conjectured that the combinators preserve the *change-executable* property as follows:

$$\begin{array}{ll}
\mathit{ChangeExec}^r\ \mathit{sf}_1\ s \wedge \mathit{ChangeExec}^r\ \mathit{sf}_2\ (\mathit{sf}_1\ s) & \Rightarrow \mathit{ChangeExec}^r\ (\mathit{sf}_1 \ggg \mathit{sf}_2)\ s \\
\mathit{ChangeExec}^r\ \mathit{sf}_1\ s \wedge \mathit{ChangeExec}^r\ \mathit{sf}_2\ s & \Rightarrow \mathit{ChangeExec}^r\ (\mathit{sf}_1\ \&\&\&\ \mathit{sf}_2)\ s \\
\mathit{ChangeExec}^r\ \mathit{sf}\ s & \Rightarrow \mathit{ChangeExec}^r\ (\mathit{freeze}\ \mathit{sf})\ s
\end{array}$$

$$\begin{array}{ll}
\mathit{NotSwitched}\ \mathit{sf}\ s \quad \wedge \mathit{ChangeExec}^r\ \mathit{sf}\ s & \Rightarrow \mathit{ChangeExec}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s \\
\mathit{Switched}\ (t_e, a)\ \mathit{sf}\ s \wedge (\lambda\ t \rightarrow \mathit{ChangeExec}^r\ (f\ a)\ (\mathit{advance}\ t_e\ s)\ (t - t_e)) & \Rightarrow \mathit{ChangeExec}^r\ (\mathit{switch}\ \mathit{sf}\ f)\ s
\end{array}$$

8.5.3 Interaction between Optimisations and Switching

In Section 8.3.1, Step and Continuous signals were defined to be *changing* at time_0. This may seem counter-intuitive; for example, one could argue that a constant signal never changes. However, this definition was chosen with optimisation in mind.

The reason pertains to the dynamic nature of signal function networks. Each signal function runs in its own local time frame (see Section 3.4.4). Consequently, what is time_0 to one signal function may not be to another. In particular, after a structural switch, the residual signal function will be at its local time_0, whereas the network external to the switching combinator will not (unless the structural switch occurred at the external time_0). Consider the case when the output is a constant Step signal. The initial value of that signal appears as a change to the rest of the network, as this is a new value that has not been seen before. If this value was considered to be *unchanging*, then the network could be incorrectly optimised based on the assumption that the value of the signal is the same as it was previously.

8.5.4 Testing Optimisations

While many of the optimisations discussed in this section appear in one form or another in many existing FRP variants, they have yet to be tested in a systematic manner on a realistic implementation of N-ary FRP. This is the obvious next step of this work. In particular, the effectiveness of *dynamic* optimisations is hard to judge without experimentation, as the overhead of applying such optimisations does not always outweigh the benefits (as experience has shown with Yampa [90]). Furthermore, the effectiveness of optimisations can be very application specific. For example, event-heavy applications tend to gain more from change-based optimisations than applications with lots of continuous-time signals and integration [90].

It seems likely that the biggest gains from these optimisations would come from the elimination of dynamic combinators, as they would then complement other techniques such as causal-commutative-arrow normalisation which are obstructed by such combinators [75]. However, investigating this remains for future work.

8.6 Conclusions

This chapter identified and formally defined several temporal properties of signals and signal functions pertaining to change and change propagation. These properties hold in the N-ary FRP model, and they would be expected to hold in any implementation that would be considered "faithful" to the semantics. Optimisation techniques for FRP implementations were also discussed, along with which properties have to hold for those optimisations to be valid.

Many of the properties defined were very similar to each other, with the differences being small and not always intuitive. A formal vocabulary to describe these notions precisely is thus a worthwhile tool in its own right.

Reasoning about change in the setting of FRP is challenging due to structural dynamism and changes due just to time passing. It is very easy to introduce invalid optimisations by failing to appreciate subtle aspects of the semantics. Having a formal framework that allows optimisation opportunities to be identified and properly justified is thus a useful aid for FRP implementers.

Chapter 9

Extensions to N-ary FRP

This chapter considers several extensions to the N-ary FRP conceptual model and type system. Specifically:

- Allowing events and changes to occur *immediately after* a time point (rather than *at* a time point);
- Refining the N-ary FRP type system to allow for safe uninitialised signals (signals that are undefined at $time_0$);
- Refining the type system of N-ary FRP with Feedback to more precisely track instantaneous dependencies between signals.

9.1 Occurrences Immediately After a Point

The N-ary FRP conceptual model (Section 4.1) only allows events to occur *at* points in time, not *immediately after* points in time. This is not the only option. The main alternatives are allowing events (and hence structural switches) to occur both at and immediately after time points, or to only allow them immediately after time points. FRP variants have made different decisions in this regard; see Daniels [32, Chapter 5] for a discussion of this.

Step signals are similar to Event signals in the N-ary FRP model in this regard, as they only assume new values *at* time points. That is, the value of a Step signal at the moment of change is that of the signal henceforth, not that of the signal hitherto. The alternatives are defining the value at the moment of change to be that of the signal hitherto, or allowing two types of change to cater for both situations.

Note that Continuous signals are less constrained, and can assume new values both at and immediately after time points.

In many cases, it is natural for an event to occur immediately after a time point. Consider an event condition such as $t > 5$; this should cause an event *immediately after* time 5, but *at* time 5 the event should not yet have occurred. This is not supported by the N-ary FRP model, and consequently the conceptual definition of the *when* signal function prohibits such event conditions (see Appendix B.7).

This is a limitation of the N-ary FRP model, and a natural solution would be to extend the model to accommodate events and changes immediately after a time point. However, extending the model in such a way has proved challenging, and is the subject of ongoing work. This section further motivates such an extension by describing some additional FRP primitives that it would accommodate, and then overviewing the issues that arise when attempting to define such an extension.

9.1.1 Additional Primitives

There are two noteworthy families of FRP primitives that the N-ary FPR model is unable to accommodate: *infinitesimal delays* and *decoupled switching combinators*. These are commonly used primitives that appear in many FRP languages, yet cannot be expressed within the N-ary FRP model.

Infinitesimal Delays

Recall the primitives *fromS* and *dfromS* (Section 4.2.4), which both coerce a Step signal to a Continuous signal. They differ only in the value they assign to the resultant Continuous signal at the moments of change in the Step signal: *fromS* defines it to be that of the signal henceforth, *dfromS* defines it to be that of the signal hitherto.

The meaning of a Step signal is that its value at the moments of change is that of the signal henceforth. Thus *fromS* can be considered to preserve the meaning of a signal, whereas *dfromS* can be considered to change the meaning of a signal by delaying it by an infinitesimal amount of time. Consequently, *dfromS* is really two distinct operations: coercion and an infinitesimal delay. Yet, in the N-ary FRP model, it is necessary to combine the two operations into a single primitive. This is because the model does not allow an infinitesimal delay to be applied (in isolation) to either a Step signal or a Continuous signal. In the former case, this is because a Step signal only allows changes *at* time points. An infinitesimal delay applied to such a signal would produce a Step signal that changes *immediately after* time points, which cannot be represented. In the latter case, although a Continuous signal that only changes in value *immediately after* time points can be defined (such as the signal produced by *dfromS*), there is no way to apply an infinitesimal delay to an *arbitrary* Continuous signal.

If the N-ary FRP model were extended to allow Step signals with changes immediately after a time point, then a primitive that solely delays a Step signal by an infinitesimal amount would be definable[1]:

$iPreS\ :\ A \to SF\ (\mathsf{S}\ A)\ (\mathsf{S}\ A)\ \mathsf{dec}$

Like *dfromS* and *delayS*, this primitive takes an initial value as an argument to define the output at $time_0$ (hence the 'i' prefix).

A similar primitive that delays an Event signal by an infinitesimal amount could also be provided:

$preE\ :\ SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)\ \mathsf{dec}$

In this case no initial value is required, as there will just not be an event occurrence at $time_0$.

[1]Signal functions that introduce an infinitesimal delay are often called *pre* for "previous".

These primitives are frequently used in FRP languages as the *decoupled* signal function within a feedback loop. For example, the *saveResume* combinator (Section 7.4.1) uses *dhold* (Section 4.3.2), which is defined using *dfromS*, to decouple the feedback loop. However, note that it would be more natural to define *dhold* using *iPreS* rather than *dfromS*. The coercion to a Continuous signal is only necessary because there is no other way to apply an infinitesimal delay to a signal in the N-ary FRP model.

Decoupled Switching Combinators

Recall the *dswitch* combinator from Section 3.4.4. It is the same as *switch*, except that the output at the moment of switching is that of the subordinate signal function rather than that of the residual signal function. This is an example of a *decoupled switching combinator* (decoupled switch), which are so called because their output signals are temporally decoupled from the Event signal that controls the switch[2]. That is, when an event occurs in this signal, the effects of this will not be observable in the output signals until immediately afterwards (even though the switch occurs, and the residual signal function starts, immediately). The *dswitch* combinator cannot be expressed in the N-ary FRP model, because it requires the capability to express Step signals that change immediately after a time point (though a more restricted version specialised to Continuous or Event signals *is* definable). Specifically, the problem is that a decoupled variant of the *splice* function (see Appendix B.5.2), which would define the value of the resultant signal at the splice time to be that of the first signal, cannot be defined. (This is the reason that a non-reflexive variant of the *change-executable* property could not be expressed in Section 8.5.2, as that required such a *splice* variant.)

Decoupled switches are particularly useful because the temporal decoupling of the output from the Event signal can be exploited when defining feedback [27]. This can be seen in the type of the *replace* combinator (and its decoupled variant, *dreplace*):

$$\begin{aligned} &\mathit{replace} &&: \mathit{SF}\ \mathit{as}\ \mathit{bs}\ d_1 \rightarrow (A \rightarrow \mathit{SF}\ \mathit{as}\ \mathit{bs}\ d_2) \rightarrow \mathit{SF}\ (\mathit{as}, \mathsf{E}\ A)\ \mathit{bs}\ \mathsf{cau} \\ &\mathit{dreplace} &&: \mathit{SF}\ \mathit{as}\ \mathit{bs}\ d_1 \rightarrow (A \rightarrow \mathit{SF}\ \mathit{as}\ \mathit{bs}\ d_2) \rightarrow \mathit{SF}\ (\mathit{as}, \mathsf{E}\ A)\ \mathit{bs}\ (d_1 \wedge d_2) \end{aligned}$$

The *replace* combinator is never decoupled, whereas the *dreplace* combinator is decoupled if its component signal functions are both decoupled. However, the type system is not precise enough to distinguish between *switch* and *dswitch*:

$$\begin{aligned} &\mathit{switch} &&: \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A)\ d_1 \rightarrow (A \rightarrow \mathit{SF}\ \mathit{as}\ \mathit{bs}\ d_2) \rightarrow \mathit{SF}\ \mathit{as}\ \mathit{bs}\ (d_1 \wedge d_2) \\ &\mathit{dswitch} &&: \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{E}\ A)\ d_1 \rightarrow (A \rightarrow \mathit{SF}\ \mathit{as}\ \mathit{bs}\ d_2) \rightarrow \mathit{SF}\ \mathit{as}\ \mathit{bs}\ (d_1 \wedge d_2) \end{aligned}$$

This will be addressed in Section 9.3.3.

9.1.2 Unresolved Issues

Attempting to extend the N-ary FRP model such that it can accommodate event occurrences and changes immediately after a time point has given rise to several unresolved issues. Many of these stem from one question: in any given signal, can there be an occurrence at a time point

[2]The term *decoupled* is (unfortunately) overloaded onto several related concepts. The three main uses are: a signal is *temporally decoupled* from another signal if it does not instantaneously depend on it; a signal function is *decoupled* if all of its output signals are temporally decoupled from all of its input signals; a *decoupled switch* is a switching combinator such that the output signal vector is temporally decoupled from the Event signal that controls the switch. Thus a *decoupled switch* is not the same as a switching combinator that constructs a *decoupled* signal function.

and then again immediately after that time point, or can there only be an occurrence at one of the two? (Note that when the same argument is relevant for both Step and Event signals, the term *occurrence* is used to subsume event occurrences and changes in Step signals.) This section considers this question, and the problems that arise from both possible answers.

No Consecutive Occurrences

First consider a model that allows occurrences immediately after a time point, but not both at and immediately after the same point. One way to formulate such a model would be as follows:

```
data When (A : Set) : Set where
  now  : A → When A
  soon : A → When A

ChangePrefix : Set → Set
ChangePrefix A = Time → ChangeList (When A)

SigVec : SVDesc → Set
SigVec (C A)    = Time → A
SigVec (E A)    = Maybe (When A) × ChangePrefix A
SigVec (S A)    = A × ChangePrefix A
SigVec (as, bs) = SigVec as × SigVec bs
```

That is, each occurrence happens either now (at the time point) or soon (immediately after the time point), but there must be a time delta between each occurrence. This will be referred to as the *Now-Soon* model.

Now, consider a signal function such as *merge* or *mapS2* that merges together two Event or Step signals. At any time t, it is possible there may be a now occurrence in one input signal, and a soon occurrence in the other. However, the output signal cannot contain both a now and soon occurrence at time t. There are thus three options: merge the occurrences, eliminate one of the occurrences, or delay one of the occurrences. Intuitively, merging the occurrences seems the correct thing to do. However, this is not possible without violating causality.

Why is this? Well, for a signal function to be *causal* its output at time t must not depend on input immediately after time t. Thus, if there is a now occurrence at time t, then the signal function must determine whether to output a now occurrence at time t, and what value to give that occurrence, before knowing if there is a soon occurrence at time t. Consequently, merging the two occurrences would be acausal. Likewise, discarding or delaying the now occurrence based on the existence of the soon occurrence would also be acausal.

The remaining options are to discard or delay the soon occurrence. If it is discarded, then information is lost. For an Event signal, event occurrences are being lost along with the values they carry. For a Step signal, the output signal is not being updated for a period of time (specifically, until the next input change). On the other hand, if the soon occurrence is delayed, then by what time delta should it be delayed? Whatever value is chosen would be arbitrary, and would introduce inaccuracy into the model. Furthermore, if the time delta chosen is larger than the (as yet unknown) time delta before the next occurrence, then the resultant output signal could be significantly corrupted.

Arguably, this could be partially addressed for Event signals by providing only a more limited version of the *merge* signal function that only emits unit events:

```
mergeUnit : SF (E A, E B) (E Unit)
```

In this case, a now event and a soon event could be merged into a single now event, with no violation of causality. No additional information is lost when this happens, albeit only because the values are *always* discarded. However, this approach would be of no use for Step signals, as a unit Step signal is completely devoid of information content.

Furthermore, the *merge* signal function is not the only primitive that can cause now and soon occurrences at the same time point. When switching, the subordinate and residual signals are spliced together temporally. The subordinate signal could contain an occurrence now, and the residual signal could contain an occurrence soon. When splicing the two signals together, there would again be the problem of how to merge the two occurrences. Taking the *mergeUnit* approach would require there to be only *unit* Event signals in the entire system!

Soon Soon is Soon Now

From the preceding discussion, it seems that occurrences must be allowed both now and soon at the same time point. The Now-Soon model is thus modified as follows:

```
data When (A : Set) : Set where
  now     : A     → When A
  soon    : A     → When A
  nowSoon : A → A → When A
SigVec (E A) = Maybe (When A) × ChangePrefix A
SigVec (S A) = A × Maybe A × ChangePrefix A
```

The difficulty of merging signals is now resolved, as merging a now occurrence and a soon occurrence simply results in a nowSoon occurrence.

The next question is whether there can be occurrences immediately after a soon occurrence, or whether that would just be the same as a soon occurrence. As a concrete example, consider the *ipreS* and *preE* signal functions. These signal functions map now occurrences to soon occurrences, but what should they do with soon occurrences? One could argue that *immediately after immediately after* t is the same as *immediately after* t, and thus that soon occurrences should be mapped to soon occurrences. Alternatively, one could argue that they are distinct, in much the same way that now and soon are distinct. Another way of putting the question is are two infinitesimal delays equal to one infinitesimal delay? The remainder of this subsection considers treating them as equal, and the next subsection considers treating them as distinct.

Assuming that *ipreS* and *preE* map soon occurrences to soon occurrences, what should they do with nowSoon occurrences? For Step signals, one could argue that the now occurrence should be discarded, leaving just a soon occurrence, as all that is of interest is its value at the time point and its value henceforth, not its value over some empty interval between the two. However, Events signals are trickier as event occurrences should not be lost. One solution would be to provide a merging function as a parameter to *preE*, for example:

```
ePreMerge : (A → A → A) → SF (E A) (E A)
```

However, this seems dubious and is certainly counter-intuitive. Inserting an infinitesimal delay in an Event signal should not cause some event occurrences therein to merge together.

Matters get worse when structural dynamism is considered. Consider a soon event triggering a structural switch, with the residual signal function producing both a now and soon occurrence at its local $time_0$. How should those occurrences correspond to the triggering event? Intuitively, the residual now occurrence would seem to correspond to the external soon event. This would mean that the local soon occurrence would be immediately after the external soon event, which is considered the same time point as the external soon. This presents two major problems. First, the residual soon occurrence occurs after the residual now occurrence in the local time frame, but they appear to occur at the same time point in the external time frame. This is not modular. Second, there then needs to be a way of merging the two occurrences, which, as previously discussed, is not possible without losing information.

An alternative would be to decide that a residual signal function always starts *at* a time point. That is, even if the triggering event occurs soon, the residual signal function starts now. In order to maintain causality, the initial now outputs of the residual signal function would be discarded. The local soon would then correspond to the external soon. However, this is not modular either. Delaying the triggering event by an infinitesimal amount of time should only have the effect of delaying the structural switch (and the output of the residual signal function) by an infinitesimal amount of time; it should not eliminate some of the residual signal function's output.

Ordered Consecutive Occurrences

Now consider immediately after soon to be distinct from soon. Once this is done, it quickly becomes apparent that it is necessary to allow for an arbitrary number of consecutive ordered occurrences immediately after a time point. Intuitively, this is because any occurrence can be infinitesimally delayed until immediately afterwards.

This is similar to a *super-dense* model of time: a model that allows a finite number of occurrences at any given time point, but assigns a chronological ordering between all occurrences that share the same point [18, 81, 84, 132]. Such a model typically defines occurrences by tagging them with two temporal co-ordinates: a point in time, and a natural number ordering the occurrences at that time point.

The proposed extension to the Now-Soon model differs from such a super-dense model in that a super-dense model considers all occurrences to be *at* a time point, whereas the Now-Soon model would consider at most one occurrence to be *at* the time point, and the remainder to be *immediately after* the time point. The important property that a signal only contains a finite number of occurrences in a finite amount of time still holds, because of the restriction that there can only be a finite number of occurrences at any given time point. The natural number that orders the occurrences would also, in the Now-Soon model, count the infinitesimal delays between occurrences. This extended Now-Soon model could be expressed as follows (where $\mathbb{N}^+$ denotes strictly positive natural numbers):

$$\begin{aligned}
&\mathit{ChangePrefix} : \mathit{Set} \to \mathit{Set}\\
&\mathit{ChangePrefix}\ A = \mathit{Time} \to \mathit{ChangeList}\ (((\Delta t \times \mathbb{N}) \uplus \mathbb{N}^+) \times A)
\end{aligned}$$

$$\begin{aligned}
&\mathit{SigVec} : \mathit{SVDesc} \to \mathit{Set}\\
&\mathit{SigVec}\ (\mathsf{C}\ A) &&= \mathit{Time} \to A\\
&\mathit{SigVec}\ (\mathsf{E}\ A) &&= \mathit{Maybe}\ A \times \mathit{ChangePrefix}\ A\\
&\mathit{SigVec}\ (\mathsf{S}\ A) &&= A \times \mathit{ChangePrefix}\ A\\
&\mathit{SigVec}\ (as, bs) &&= \mathit{SigVec}\ as \times \mathit{SigVec}\ bs
\end{aligned}$$

That is, there is a gap between each occurrence of either a time delta and a natural number of infinitesimal delays, or a strictly positive natural number of infinitesimal delays.

There is no longer any need to merge occurrences when applying infinitesimal delays: the number of infinitesimal delays is just incremented. When merging two signals, occurrences at the same time point are merged only if the numbers of infinitesimal delays are equal; otherwise they remain two separate occurrences.

This approach seems promising, but the ramifications of this model on the N-ary FRP primitives need to be considered. This remains as future work.

Interaction with Continuous Signals

However, there is a further problem concerning the interaction between `soon` occurrences (or super-dense occurrences) and continuous signals. For example, the *sampleWithC* signal function merges an Event and Continuous signal by sampling the Continuous signal at the time points of the event occurrences, and combining that value with the value of the event. For a `soon` event, what value from the Continuous signal should be combined with it? A continuous signal is a function from time to value, it cannot take "immediately after" a time point as an argument. The value of the Continuous signal at the time point could be used, but that would be incorrect for a Continuous signal that changes immediately after a point (consider an event occurring immediately after time x for the Boolean Continuous signal $(\lambda\ t\ \rightarrow\ t > x)$).

9.1.3 Summary and Related Work

The N-ary model is limited by events that can only occur at, not immediately after, points in time. Similarly, it has Step signals that change at, not immediately after, time points. This means that the N-ary FRP model cannot describe common FRP primitives such as infinitesimal delays and decoupled switching combinators. It would seem natural to extend the N-ary FRP model to accommodate such functionality, but it is unclear how this should be done. As discussed, a super-dense model of time seems the most promising approach, but investigating this is future work.

The problem is addressed in a variety of ways in other FRP models. In discrete-time models (or discrete-time implementations of continuous-time models) the problem does not arise, as *immediately after* a time point is just the next time sample. Other continuous-time models of FRP have either disallowed infinitesimal delays [65, 127], or postulated the existence of an infinitesimal time delta for use in expressing the semantics [25]. Decoupled switching combinators are found in most FRP variants; indeed, in some FRP variants *all* switches are decoupled switches. However, most FRP variants do not model Step signals as a distinct signal kind, which is a sufficient restriction to allow decoupled switches to be defined (as discussed in Section 9.1.1). A notable exception is Reactive, which deals with the problem by *only* allowing Step signals to change immediately after a time point [37] (although it is then impossible to define non-decoupled switches).

9.2 Type-safe Initialisation

In many FRP and synchronous data-flow languages (e.g. Yampa [92], Signal [68] , Lustre [48] and Lucid Synchrone [104]), it is possible to define signals that are undefined at $time_0$.

These are called *uninitialised signals* because they lack an initial value. Without such signals, a programmer would have to provide a "dummy" initial value for signals in situations where the initial value is not used. This makes the language more expressive (as situations where no such value is available would otherwise be inexpressible), and also makes FRP programs clearer as dummy values can obfuscate code [21].

The disadvantage of uninitialised signals is that they can cause run-time errors in an implementation if their (undefined) initial value *is* used. Some languages leave the correct use of uninitialised signals as a responsibility of the programmer (e.g. Yampa), while others perform static checks (e.g. Lucid Synchrone). Such checks tend to be conservative in nature, often requiring a signal to be initialised several times [21]. Some languages simply do not allow uninitialised signals at all, as is the case with N-ary FRP.

This section defines *N-ary FRP with Uninitialised Signals*, an extension of N-ary FRP that *does* allow uninitialised signals. This is achieved through a type-system refinement that ensures undefined initial values are never used in a well-typed program. The type system is similar in style to that of Colaço and Pouzet [21]; the differences are discussed in Section 9.2.5. Introducing uninitialised signals is orthogonal to introducing feedback, and so N-ary FRP, rather than N-ary FRP with Feedback, is taken as the base language for simplicity. However, as shown in Sculthorpe and Nilsson [112], combining feedback and uninitialised signals is both natural and straightforward.

9.2.1 Infinitesimal Delays

Many FRP and synchronous data-flow languages provide a primitive that delays a signal by some minimal amount:

$pre : Signal\ A \rightarrow Signal\ A$

In a sampled implementation, this minimal amount is one time step. For languages with a discrete-time conceptual model, this is a very basic operation. For languages with a continuous-time model, this is generally expressed as an infinitesimal delay in the signal (as discussed in Section 9.1.1). There are two important properties about such a primitive: it is *decoupled*, and it produces an *uninitialised* signal.

Languages providing *pre* also provide a corresponding primitive to initialise such uninitialised signals:

$initialise : A \rightarrow Signal\ A \rightarrow Signal\ A$
$initialise\ a\ s \approx \lambda\ t \rightarrow \mathbf{if}\ s > 0\ \mathbf{then}\ s\ t\ \mathbf{else}\ a$

That is, *initialise* overwrites the initial value of a signal with the provided value[3]. Note that in a multi-kinded FRP model there is no need to initialise event signals: an uninitiated event signal just corresponds to an event signal with no occurrence at $time_0$.

Languages that disallow uninitialised signals often provide a single primitive that introduces an infinitesimal delay and initialises the resultant signal, effectively combining *pre* and *initialise*:

$iPre : A \rightarrow Signal\ A \rightarrow Signal\ A$
$iPre\ a\ s \approx initialise\ a\ (pre\ s)$

[3]In many languages *initialise* is denoted −>.

This is the approach taken by N-ary FRP. The *dfromS* primitive corresponds to an instance of *iPre* specialised to an input Step signal and an output Continuous signal. Likewise, the *iPreS* primitive discussed in Section 9.1.1 is the instance specialised to Step signals.

However, there are several reasons why combining *pre* and *initialise* into a single primitive is limiting:

- It may be inconvenient, or impossible (if no initial value is available), to initialise the signal at the usage of *pre*, compared to elsewhere in the program.
- A lifted pure function can be applied to an uninitialised signal without causing an error, it just produces an uninitialised output signal.
- Some primitives do not require their input signals to be initialised, yet still produce initialised output.

The remainder of this section refines the type system of N-ary FRP to allow uninitialised Continuous signals. Specifically, signal vector descriptors are modified such that Continuous signals are tagged as being either initialised or uninitialised. A similar refinement should be valid for Step signals; but, as discussed in Section 9.1, N-ary FRP does not yet allow Step signals that change immediately after a point (which is required to express the signal becoming initialised immediately after $time_0$). Nevertheless, Continuous signals are sufficient to demonstrate the interesting aspects of the type system.

9.2.2 Initialisation Descriptors

First, a data type of *initialisation descriptors* is introduced. These are tags that will be included in signal vector descriptors to describe the initialisation properties of signals.

```
data Init : Set where
  ini : Init   -- initialised signal
  uni : Init   -- uninitialised signal
```

As only uninitialised *Continuous* signals are supported by this extension, signal vector descriptors are refined as follows:

```
data SVDesc : Set where
  C    : Init → Set → SVDesc
  E    : Set → SVDesc
  S    : Set → SVDesc
  _,_  : SVDesc → SVDesc → SVDesc
```

Signal vectors are then refined:

```
SigVec : SVDesc → Set
SigVec (C ini A)  = Time → A
SigVec (C uni A)  = Time⁺ → A
SigVec (E A)      = Maybe A × ChangePrefix A
SigVec (S A)      = A × ChangePrefix A
SigVec (as, bs)   = SigVec as × SigVec bs
```

That is, they are defined as before (Section 4.1.4) except that a Continuous signal may be *uninitialised*, in which case it is not defined at $time_0$.

Finally, three utility functions over these modified descriptors are defined:

$$
\begin{array}{lcl}
\mathit{iniSV} & : & \mathit{SVDesc} \rightarrow \mathit{SVDesc} \\
\mathit{iniSV}\ (\mathsf{C}\ \_\ A) & = & \mathsf{C}\ \mathsf{ini}\ A \\
\mathit{iniSV}\ (\mathsf{E}\ A) & = & \mathsf{E}\ A \\
\mathit{iniSV}\ (\mathsf{S}\ A) & = & \mathsf{S}\ A \\
\mathit{iniSV}\ (as, bs) & = & (\mathit{iniSV}\ as, \mathit{iniSV}\ bs)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{uniSV} & : & \mathit{SVDesc} \rightarrow \mathit{SVDesc} \\
\mathit{uniSV}\ (\mathsf{C}\ \_\ A) & = & \mathsf{C}\ \mathsf{uni}\ A \\
\mathit{uniSV}\ (\mathsf{E}\ A) & = & \mathsf{E}\ A \\
\mathit{uniSV}\ (\mathsf{S}\ A) & = & \mathsf{S}\ A \\
\mathit{uniSV}\ (as, bs) & = & (\mathit{uniSV}\ as, \mathit{uniSV}\ bs)
\end{array}
$$

$$
\begin{array}{lcl}
\_\sqcap\_ & : & \mathit{Init} \rightarrow \mathit{Init} \rightarrow \mathit{Init} \\
\mathsf{ini} \sqcap \mathsf{ini} & = & \mathsf{ini} \\
\_\ \ \sqcap\ \_ & = & \mathsf{uni}
\end{array}
$$

The *iniSV* and *uniSV* functions set all initialisation descriptors to be initialised or uninitialised, respectively. The $\sqcap$ operator is conjunction of initialisation descriptors.

9.2.3 Subtyping

Initialised signals are a subtype of uninitialised signals, as they can always be substituted in their place (by "forgetting" the initial value of the signal). In a language without subtyping, an explicit weakening primitive could be provided to coerce initialised signals to uninitialised signals. To express such a primitive, subtyping relations on initialisation and signal vector descriptors are required:

$$
\begin{array}{lcl}
\_\langle:\_ & : & \mathit{Init} \rightarrow \mathit{Init} \rightarrow \mathit{Set} \\
\mathsf{uni}\ \langle:\ \mathsf{ini} & = & \mathit{False} \\
\_\ \ \langle:\ \_ & = & \mathit{True}
\end{array}
$$

$$
\begin{array}{llcl}
\_<:\_ & : \mathit{SVDesc} \rightarrow \mathit{SVDesc} \rightarrow \mathit{Set} & & \\
\mathsf{C}\ i_1\ A & <: \mathsf{C}\ i_2\ B & = & (i_1 \langle:\ i_2) \times (A \equiv B) \\
\mathsf{E}\ A & <: \mathsf{E}\ B & = & A \equiv B \\
\mathsf{S}\ A & <: \mathsf{S}\ B & = & A \equiv B \\
(as_1, bs_1) & <: (as_2, bs_2) & = & (as_1 <: as_2) \times (bs_1 <: bs_2) \\
\_ & <: \_ & = & \mathit{False}
\end{array}
$$

A primitive weakening combinator can then be given the following type:

$$\mathit{weaken}\ :\ as' <: as \rightarrow bs <: bs' \rightarrow \mathit{SF}\ as\ bs \rightarrow \mathit{SF}\ as'\ bs'$$

9.2.4 Refined Primitives

This section gives the refined types of the N-ary FRP primitives. Only those that require modification are considered. Note that some of these types could be more polymorphic, which would make programming with them more convenient in a host language without subtyping. That is not done so here to avoid complicating the presentation.

Primitive Signal Functions

First, *initialise* is added as an additional primitive:

$$\mathit{initialise}\ :\ A \rightarrow \mathit{SF}\ (\mathsf{C}\ \mathsf{uni}\ A)\ (\mathsf{C}\ \mathsf{ini}\ A)$$

Instead of requiring an initial value, *dfromS* now produces an uninitialised signal:

$$\mathit{dfromS}\ :\ \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{C}\ \mathsf{uni}\ A)$$

Whereas *fromS* produces an initialised signal:

$fromS \;:\; SF\;(\mathsf{S}\;A)\;(\mathsf{C}\;\mathsf{ini}\;A)$

Integration always produces an initialised signal (the output at $time_0$ is always 0), even if the input signal is uninitialised:

$integralC \;:\; SF\;(\mathsf{C}\;i\;\mathbb{R})\;(\mathsf{C}\;\mathsf{ini}\;\mathbb{R})$
$integralS \;:\; SF\;(\mathsf{S}\;\mathbb{R})\;(\mathsf{C}\;\mathsf{ini}\;\mathbb{R})$

Delaying a Continuous signal always produces an initialised signal, as the initialisation function provides the initial value. If the input signal is uninitialised, then the initialisation function will also provide the output at the time point equal to the delay time.

$delayC \;:\; Time^{+} \to (Time \to A) \to SF\;(\mathsf{C}\;i\;A)\;(\mathsf{C}\;\mathsf{ini}\;A)$

The *when* signal function does not require initialised input, as it never produces an event at $time_0$:

$when \;:\; (A \to Bool) \to SF\;(\mathsf{C}\;i\;A)\;(\mathsf{E}\;A)$

Lifting Functions

The lifting functions produce initialised signals if all of their input signals are initialised:

$liftC \;:\; (A \to B) \to SF\;(\mathsf{C}\;i\;A)\;(\mathsf{C}\;i\;B)$
$liftC2 \;:\; (A \to B \to Z) \to SF\;(\mathsf{C}\;i_1\;A, \mathsf{C}\;i_2\;B)\;(\mathsf{C}\;(i_1 \sqcap i_2)\;Z)$
$sampleWithC \;:\; (A \to B \to Z) \to SF\;(\mathsf{C}\;i\;A, \mathsf{E}\;B)\;(\mathsf{E}\;Z)$

In the case of *sampleWithC*, this means an initial event occurrence will be discarded if the Continuous signal is uninitialised.

Switch

Recall that each signal function exists in its own local time frame (Section 3.4.4). When a residual signal function is switched in, its local $time_0$ will not be $time_0$ from the perspective of the external network[4]. Consequently, if a residual signal function produces an uninitialised signal, it could cause a signal to be undefined at some arbitrary time point in the external network. This could be disastrous. To avoid this, the type of *switch* requires the residual signal function to produce initialised signals:

$switch \;:\; SF\;as\;(bs, \mathsf{E}\;A) \to (A \to SF\;as\;(iniSV\;bs)) \to SF\;as\;bs$

However, this is not the case for decoupled variants of switching combinators (introduced in Section 9.1.1). The initial output from the residual signal function of a decoupled switch is never used, and thus the problem of an undefined value escaping its local time frame does not arise. Indeed, the output of the residual signal function of *dswitch* can be entirely uninitialised:

$dswitch \;:\; SF\;as\;(bs, \mathsf{E}\;A) \to (A \to SF\;as\;(uniSV\;bs)) \to SF\;as\;bs$

In both cases there is no need to modify the input type of the residual signal function, as the residual signal function can just ignore defined input if it is expecting it to be undefined.

[4]Except in the case of the switch occurring at $time_0$.

Freeze

For the *freeze* combinator, the Continuous signal of “frozen” signal functions is initialised. But care must be taken with the *type* of the frozen signal function. Consider: a signal function must never receive undefined input, except at $time_0$. A frozen signal function has already been running for some amount of time, and thus (unless that amount of time is zero) is not at its local $time_0$ when switched-in again. Thus, even if the subordinate signal function accepts uninitialised signals, the frozen signal function must not do so lest it receive an undefined value after its local $time_0$. The *freeze* combinator is therefore refined as follows:

$freeze : SF\ as\ bs \rightarrow SF\ as\ (bs, \mathsf{C}\ \mathsf{ini}\ (SF\ (iniSV\ as)\ bs))$

Note that the output type of the frozen signal function does not need to be modified. If the switched-in signal function produces initialised output in a context that expects uninitialised signals, then the initial values can simply be discarded.

9.2.5 Summary and Related Work

This section extended the N-ary FRP language with uninitialised signals. The type system ensures that undefined initial values are never used. The semantics of the modified primitives were omitted as they were not crucial to the discussion. Their Agda encoding is available in the online archive [1].

The refined type system is similar to that of Colaço and Pouzet [21], which addresses the same problem in the context of the synchronous data-flow languages Lustre and Lucid Synchrone. There are two main differences between this work and theirs. First, they only consider static networks, whereas N-ary FRP allows dynamism. Second, theirs is a single-kinded setting, whereas N-ary FRP is multi-kinded. Multi-kinded signals allow for greater precision, such as expressing that Event signals may always be uninitialised.

9.3 Decoupledness Matrices

Chapter 7 introduced a type system for N-ary FRP with Feedback that rules out ill-defined feedback. However, that type system is conservative, rejecting some programs that only contain well-defined feedback. This section considers a further refinement of the type system that is more permissive in the feedback it will accept.

9.3.1 Motivation

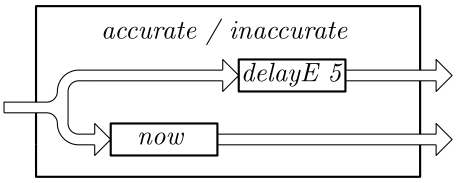
In Section 7.3.3 it was observed that a *loop′* combinator can be defined, but that the type system assigns the composite signal function the index cau when it should be assigned dec. For a simpler example of the same issue, consider the following two signal functions:

$accurate : SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A, \mathsf{E}\ Unit)\ \mathsf{dec}$
$accurate = (identity \ggg delayE\ 5)\ \&\&\&\ (now \ggg identity)$

$inaccurate : SF\ (\mathsf{E}\ A)\ (\mathsf{E}\ A, \mathsf{E}\ Unit)\ \mathsf{cau}$
$inaccurate = (identity\ \&\&\&\ now) \ggg (delayE\ 5 *\!*\!* identity)$

Both signal functions represent the same network (see Figure 9.1), yet one is typed as decoupled while the other is typed as causal. The problem is that the decoupledness indices do not contain

Figure 9.1 The network underlying *accurate* and *inaccurate*



all decoupledness information, merely a conservative approximation of it. Thus, depending on the order in which routing combinators are applied, different amounts of information is lost. For example, *identity* &&& *now* is typed as causal, and that *now* is decoupled is forgotten. Or, more precisely, it is forgotten that the second output signal is temporally decoupled from the input signal.

Ideally, the decoupledness index of a signal function network should depend on the semantics of the routing primitives used, not the particular way those routing primitives are used to express its structure. The type-system refinement in the next section addresses this by ensuring that the routing primitives (and dynamic combinators) precisely track which signals are temporally decoupled from which.

9.3.2 Type System

N-ary FRP with Feedback indexes each signal function by a *single* decoupledness value, which essentially describes whether *all* output signals are temporally decoupled from *all* input signals. This leads to imprecision, because it is impossible to express that *some* output signals do not instantaneously depend on *some* input signals. An alternative approach is to index each signal function by a *matrix* of decoupledness values, with each element of the matrix describing whether a *single* input signal is temporally decoupled from a *single* output signal. This allows for far greater precision.

Decoupledness Matrices

Familiarity with Boolean matrices and basic operations on them is assumed. Consequently, this subsection just introduces the syntax and omits the definitions.

The type *Matrix m n* denotes an $m \times n$ matrix of *Dec* values:

$$Matrix : \mathbb{N} \to \mathbb{N} \to Set$$

Horizontal and vertical matrix concatenation are denoted as follows:

$$\begin{aligned} \_{+\!\!+}_h\_ &: Matrix\ l\ m \to Matrix\ l\ n \to Matrix\ l\ (m+n) \\ \_{+\!\!+}_v\_ &: Matrix\ l\ n \to Matrix\ m\ n \to Matrix\ (l+m)\ n \end{aligned}$$

The $+$ and $*$ operators are overloaded for matrix addition and matrix multiplication, taking $\wedge$ to be the underlying additive operator and $\vee$ to be the underlying multiplicative operator:

$$\begin{aligned} \_+\_ &: Matrix\ m\ n \to Matrix\ m\ n \to Matrix\ m\ n \\ \_*\_ &: Matrix\ l\ m \to Matrix\ m\ n \to Matrix\ l\ n \end{aligned}$$

The identity matrix is denoted I, taking `dec` to be the zero element and `cau` to be the unit element:

$I \;:\; \mathit{Matrix}\; n\; n$

The zero matrix is a matrix consisting entirely of `dec` elements:

$M_{dec} \;:\; \mathit{Matrix}\; m\; n$

Finally, a function that extends a column vector horizontally by replicating that column has the following type:

$\mathit{extend} \;:\; \mathit{Matrix}\; m\; 1 \to \mathit{Matrix}\; m\; n$

Refined Signal Functions

In N-ary FRP, signal functions are parametrised on signal vector descriptors. To determine the appropriately sized matrix for a signal function, the number of signals in its signal vectors needs to be computed. This is achieved by the following function:

$$\begin{array}{lcl}
\mathit{svlength} \;:\; \mathit{SVDesc} \to \mathbb{N} & & \\
\mathit{svlength}\; (\mathsf{C}\; A) & = & 1 \\
\mathit{svlength}\; (\mathsf{E}\; A) & = & 1 \\
\mathit{svlength}\; (\mathsf{S}\; A) & = & 1 \\
\mathit{svlength}\; (as, bs) & = & \mathit{svlength}\; as + \mathit{svlength}\; bs
\end{array}$$

A matrix parametrised on signal vector descriptors can then be defined:

$$\begin{array}{l}
\mathit{SVMatrix} \;:\; \mathit{SVDesc} \to \mathit{SVDesc} \to \mathit{Set} \\
\mathit{SVMatrix}\; as\; bs \;=\; \mathit{Matrix}\; (\mathit{svlength}\; as)\; (\mathit{svlength}\; bs)
\end{array}$$

The refined signal function type is thus:

$\mathit{SF} \;:\; (as\; bs \;:\; \mathit{SVDesc}) \to \mathit{SVMatrix}\; as\; bs \to \mathit{Set}$

In this formulation, a signal function is *decoupled* if its matrix is M_{dec} (all input signals are decoupled from all output signals).

9.3.3 Retyping the Primitives

The primitive signal functions can now be retyped.

Atomic Routers

For the atomic routers, each output signal either depends on an input signal instantaneously, or it does not depend on it at all. If there is no dependency, then the signals are clearly temporally decoupled. Thus, these primitives are indexed as follows:

$$\begin{array}{lcl}
\mathit{identity} & : & \mathit{SF}\; as\; as\; I \\
\mathit{sfFst} & : & \mathit{SF}\; (as, bs)\; as\; (I \mathbin{+\!\!+}_v M_{dec}) \\
\mathit{sfSnd} & : & \mathit{SF}\; (as, bs)\; bs\; (M_{dec} \mathbin{+\!\!+}_v I)
\end{array}$$

Recall that `cau` is the unit element and `dec` is the zero element. Thus, the identity matrix (I) expresses that each output signal only depends instantaneously on its corresponding input signal, and the zero matrix (M_{dec}) expresses that all output signals are decoupled from all input signals. In this case, the zero matrix is used to express that the output signals are temporally decoupled from the discarded input signals.

Acyclic Routing Combinators

The matrix for sequential composition is defined by matrix multiplication:

$\_\ggg\_ : SF\ as\ bs\ m_1 \rightarrow SF\ bs\ cs\ m_2 \rightarrow SF\ as\ cs\ (m_1 * m_2)$

Intuitively, an output signal instantaneously depends on an input signal if there is any intermediate signal that both depends instantaneously on the input signal and is instantaneously depended on by the output signal.

The matrix for fan-out is defined by horizontal matrix concatenation:

$\_\&\&\&\_ : SF\ as\ bs\ m_1 \rightarrow SF\ as\ cs\ m_2 \rightarrow SF\ as\ (bs, cs)\ (m_1 +\!\!+_h\ m_2)$

Dynamic Combinators

In the case of the *freeze* combinator, the second output is temporally decoupled from all inputs, and the decoupledness of the first output is given by the decoupledness of its subordinate signal function:

$freeze : SF\ as\ bs\ m \rightarrow SF\ as\ (bs, \mathsf{C}\ (SF\ as\ bs\ m))\ (m +\!\!+_h\ M_{dec})$

The *switch* combinator is a little more complicated. At first glance, its decoupledness would seem to be the matrix addition of the subordinate and residual signal functions. However, note that the overall output of a *switch* depends instantaneously on the event signal, as at the moment of switching the residual signal function is switched-in and the overall output is taken from that. Thus, all output signals depend instantaneously on the event signal, and, consequently, all output signals depend instantaneously on any signals that the Event signal depends instantaneously upon. This can be expressed as follows:

$switch : SF\ as\ (bs, \mathsf{E}\ A)\ (m_1 +\!\!+_h\ m_e) \rightarrow (A \rightarrow SF\ as\ bs\ m_2) \rightarrow SF\ as\ bs\ (m_1 + m_2 + extend\ m_e)$

Section 9.1.1 introduced *decoupled switches*: switching combinators whose output at the moment of switching is that of the subordinate signal function. For these switches, the output signals do not depend instantaneously on the event (hence the name). Thus, for example, the type of a *dswitch* combinator would be:

$dswitch : SF\ as\ (bs, \mathsf{E}\ A)\ (m_1 +\!\!+_h\ m_e) \rightarrow (A \rightarrow SF\ as\ bs\ m_2) \rightarrow SF\ as\ bs\ (m_1 + m_2)$

In Yampa the *dswitch* is often used because of this stronger decoupling, even though this is not visible in its type (or checked by the compiler) and relies on the programmer to have a good understanding of what she is doing.

Feedback Combinators

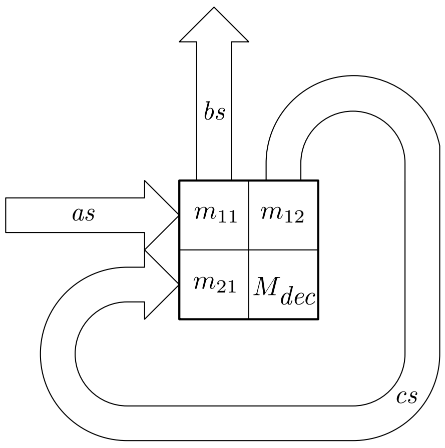
The *loop* combinator is easy to define by requiring the feedback signal function to be entirely decoupled:

$loop : SF\ (as, cs)\ bs\ (m_1 +\!\!+_v\ m_2) \rightarrow SF\ cs\ bs\ M_{dec} \rightarrow SF\ as\ bs\ m_1$

However, with this more precise decoupling, explicitly separating the feedback and feed-forward signal functions is unnecessary. Instead, the *arrowLoop* combinator could be used:

$arrowLoop : SF\ (as, cs)\ (bs, cs)\ ((m_{11} +\!\!+_v\ m_{21}) +\!\!+_h\ (m_{12} +\!\!+_v\ M_{dec})) \rightarrow SF\ as\ bs\ (m_{11} + m_{12} * m_{21})$

Figure 9.2 The decoupledness matrix for *arrowLoop*



This type may be best understood graphically (Figure 9.2). Essentially, it states that:

- the output to be fed-back cannot instantaneously depend on the fed-back input (M_{dec});
- the output to be fed-back can depend on the overall input in any way (m_{12});
- the overall output can depend on the fed-back input in any way (m_{21});
- the resultant decoupledness matrix ($m_{11} + m_{12} * m_{21}$) is the addition of the direct connections between the inputs and outputs (m_{11}), and the connections that go around the loop once ($m_{12} * m_{21}$).

Example: Inaccurate

Finally, consider the *inaccurate* signal function again. Its component signal functions are typed as follows:

$$
\begin{array}{lll}
\mathit{now} & : & \mathit{SF}\ \mathit{as}\ (\mathsf{E}\ \mathit{Unit})\ M_{dec} \\
\mathit{delayE} & : & \mathit{Time}^{+} \rightarrow \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)\ M_{dec} \\
\_{*}{*}{*}\_ & : & \mathit{SF}\ \mathit{as}\ \mathit{bs}\ m_1 \rightarrow \mathit{SF}\ \mathit{cs}\ \mathit{ds}\ m_2 \rightarrow \mathit{SF}\ (\mathit{as}, \mathit{cs})\ (\mathit{bs}, \mathit{ds})\ ((m_1 +\!\!+_v M_{dec}) +\!\!+_h (M_{dec} +\!\!+_v m_2))
\end{array}
$$

Putting them together now assigns the *inaccurate* signal function the more accurate M_{dec} index:

$$
\begin{array}{lll}
\mathit{inaccurate} & : & \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{E}\ A, \mathsf{E}\ \mathit{Unit})\ M_{dec} \\
\mathit{inaccurate} & = & (\mathit{identity}\ \&\&\&\ \mathit{now}) \ggg (\mathit{delayE}\ 5\ {*}{*}{*}\ \mathit{identity})
\end{array}
$$

9.3.4 Summary and Related Work

This section refined the type system of N-ary FRP with Feedback to more precisely track the instantaneous dependencies between signals. The idea is that the type of each signal function should record precisely which output signals are temporally decoupled from which input signals, rather than just recording whether *all* input signals are temporally decoupled from *all* outputs signals, as was the case previously. This information can be represented by a Boolean matrix, and the matrices of the primitives can be defined by basic matrix operations. This N-ary FRP variant will be referred to as *N-ary FRP with Decoupledness Matrices.*

The matrix approach was inspired by the *structural types* of Nilsson [91], which, in a setting of modular equation systems, assigns incidence matrices to the types of equation-system fragments as a means of statically detecting (structurally) over- and under-determined equations.

9.4 Conclusions

N-ary FRP with Uninitialised Signals can be embedded in Agda in a similar manner as N-ary FRP with Feedback. The code for such an embedding is available in the online archive [1], but is not included in this thesis as the modifications are mostly trivial, yet tedious and extensive, coercions between initialised and uninitialised signals. An equivalent Haskell embedding has not yet been encoded, and it remains to be seen whether the Haskell type system is up to the task.

Programming with initialisation descriptors is similar to programming with decoupledness descriptors, as they are both just type-level Booleans. Thus the same problem of Boolean expressions not β-reducing arises, and the same potential solutions are relevant (see Section 7.7).

The type system of N-ary FRP with Decoupledness Matrices has been encoded in Agda. However, defining library combinators using this type system is unpleasant. The lack of β-reduction in the type indices is a lot harder to overcome when the expressions contain Boolean matrices of unknown size, rather than just a finite number (typically no more than three) of Booleans. The required matrix equivalences can be proved in Agda, but this is too much work to be a practical approach. A type checker that can automatically solve Boolean matrix constraints would seem to be essential for this type system to be viable.

N-ary FRP with Decoupledness Matrices has not yet been implemented. The main complication in extending the Agda embedding of N-ary FRP with Feedback is that it is no longer possible to assign a single transition function to each signal function. Instead, a signal function requires a set of transition functions to cater for all possible orders of execution. For example, consider a signal function that has two input signals and two output signals, where the first output instantaneously depends on the first input (and is decoupled from the second input) and the second output instantaneously depends in the second input (and is decoupled from the first input). The first output may be required before the second input is available, or the second output may be required before the first input is available. Thus two transition functions are needed, one for each order. More generally, a transition function is needed for every possible order that the output signals can be demanded in. Identifying and implementing a minimal set of such transition functions has been the subject of recent work in the context of code generation for synchronous data-flow languages [78, 79, 106].

Chapter 10

Related Work

This chapter overviews other conceptual models of FRP, and discusses some of the safety guarantees and optimisation techniques present in FRP and other reactive languages.

10.1 Conceptual Models of FRP

Devising conceptual models for FRP is nothing new. Daniels [32] has constructed a complete formal semantics for a small Fran-inspired CFRP language called CONTROL. The semantics of CONTROL assume exact real numbers and an idealised implementation with no approximation errors. His approach is similar to that taken by this thesis: define the desired semantics first as a *basis* for implementation, rather than giving semantics to an *existing* implementation.

CONTROL is a single-kinded language: only continuous-time behaviours are a first-class abstraction. Both signals and signal generators are definable in CONTROL, but this is not distinguished in their types (they are both behaviours). Structural switching is controlled by Boolean behaviours, with the moment of switching being when the Boolean is first true. The residual behaviour does not depend on any signal value, and thus is determined in advance. Consequently CONTROL is only structurally dynamic, unlike most FRP variants which are highly structurally dynamic.

Instantaneous feedback can be expressed in CONTROL, including ill-defined feedback. However, the semantics ensure that two noteworthy classes of instantaneous feedback are well-defined. First, integration is defined in a decoupled manner, so a signal may depend instantaneously on its own integral. Second, the semantics of switching is such that whether a structural switch occurs is determined under the assumption that the switch has not occurred. The latter is useful because CONTROL's switching combinator defines its overall value at the moment of switching to be that of the residual behaviour. Thus the Boolean behaviour that controls a structural switch can instantaneously depend on the overall value of the switching combinator, without causing divergence in a situation where the value of the subordinate signal triggers a structural switch but the value of the residual signal does not.

The CFRP model defined in this thesis is based on that of Wan and Hudak [127]. The aim of their work was to show that a sampled implementation can converge to their semantics as the sampling rate tends to zero, provided certain constrains are placed on the primitives.

Table 10.1 Naming conventions for signal kinds

FRP Variant	Signal Kind		
N-ary FRP	Event Signal	Step Signal	Continuous Signal
Reactive	Event	Reactive Value	Time Function
Grapefruit	Discrete Signal	Segmented Signal	Continuous Signal

For example, only behaviours that converge uniformly can be integrated, and “spikes” in the input signal to the *when* primitive are prohibited. N-ary FRP takes a similar approach in that the semantics of some primitive signal functions are only defined for well-behaved input signals (see Section 4.4). All switching combinators in Wan and Hudak’s model are decoupled (see Section 9.1.1), and they do not consider feedback.

More recently, King [65] has defined a semantics for a small set of FRP primitives that he uses to define a subset of both the Yampa [92] and FrTime [23] implementations. King’s primitives are expressed using temporal logic, which he also uses to state and prove properties about the primitives. In particular, he is concerned with *timestep irrelevance* (the semantics of a primitive should be independent of the sampling rate used in an implementation) and *time invariance* (the semantics of a primitive should not depend on the global time, only its local time frame). These are both properties that hold in the N-ary FRP conceptual model. The Yampa and FrTime primitives that King does not define are those that depend on the sampling rate, and thus do not satisfy the timestep-irrelevance property.

In both King’s and Wan and Hudak’s models, continuous-time behaviours and discrete-time events are distinct first-class abstractions in the languages. However, unlike N-ary FRP, step signals are not considered separately. Also, along with CONTROL, these models are based around first-class signals (rather than abstract first-class signal functions), and consequently lack a notion of freezing signal functions.

The UFRP model defined in this thesis is based on the semantics of Yampa defined by Courtney [25]. However, in his semantics, event signals are defined as continuous-time signals carrying option types (corresponding directly to the Yampa implementation). This differs from the UFRP model, where the semantics of an Event signal is a finite list of occurrences up to the current time. The UFRP model is defined in this way to make the correspondence with the other models in this thesis clearer. It also allows for a *computable* semantics to be given to *switch*, which cannot be done in Courtney’s model (as the time of the first event occurrence cannot be computed) [25, Chapter 4].

Two Haskell-embedded FRP implementations currently under development are *Reactive* [37] and *Grapefruit* [63]. They both identify the three signal kinds (see Table 10.1), and use a push-based implementation for step and event signals. Reactive uses a pull-based implementation for continuous signals, whereas, at time of writing, the implementation of continuous signals is still under development in Grapefruit. Signals are first class in both systems, though Grapefruit also has a first-class signal-function abstraction for switching purposes.

Central to FRP’s hybrid capabilities is the notion of events occurring at specific points in time, and specifying reactions to such events. This means asking whether some event has occurred yet or not. A natural way of doing this is to compare the time associated with the event with the present time. However, this directly leads to a causality problem: how can the precise future time of an event that has not yet occurred be known in general? Predicating an FRP semantics on such a capability would make the whole model non-causal, severely limiting its usefulness for describing the meaning of FRP programs.

The key to resolving this dilemma is to concentrate on the original question above, whether an event has occurred yet or not, not the exact future time of its occurrence. In the original work on Fran, this was achieved through a careful definition of a customised time domain with an ordering that permitted deciding whether one time value is before another without knowing the exact value of the second [38]. The same problem is addressed in a similar way in Reactive by making events "future values". Grapefruit deals with the issue by considering all possible interleavings of future event occurrences, relying on laziness to ensure that only the correct interleaving is evaluated. In the N-ary FRP model, this problem is addressed more directly by building a notion of observation *only up to some specific point in time* into the definitions of Event and Step signals. This leads to a clear and simple semantics as it does not rely on any auxiliary notions, and also to a finitary semantics for events and changes. This approach is unlikely to be very useful as a direct basis for implementation, but the purpose of the N-ary FRP semantics is not to serve as a basis for some specific implementation, but rather to serve as a reference relevant for *any* implementation.

In CONTROL, yet another approach is taken by assigning a signal a *set* of times for which it is alive [32], rather than a specific start time or local time frame. Whether a structural switch has occurred is then determined by whether the Boolean behaviour (that controls the switch) is true for any time in the set (of times for which it is alive) up to the present, without the precise time point at which it first becomes true being required. This also allows structural switches that occur both *at* and *immediately after* a time point to be expressed, by using sets with and without a minimum element, respectively. The latter is a capacity that is lacking in the N-ary FRP model (see Section 9.1).

Elerea [99, 100] is another Haskell embedding of FRP currently in development. Elerea has first-class signals and signal generators (as distinct types), but is otherwise in many ways similar to Yampa, being a single-kinded pull-based system. In contrast to Yampa, Elerea has a discrete-time semantics that doesn't abstract away from the discrete implementation. Yampa provides a set of primitives that operate on conceptually continuous-time signals and conceptually discrete-time events, trying to hide the sampling rate from the programmer. Elerea, on the other hand, exposes the sampling rate, reducing the number of primitives required. Similarly, whereas Yampa provides an abstract event type that is internally *implemented* as an option type, Elerea directly uses signals carrying option types (or Booleans) to achieve event-like behaviour.

Dynamism is expressed in Version 2 of Elerea through a monadic *join* for signals [100]:

$join : Signal\ (Signal\ A) \rightarrow Signal\ A$

The same approach is taken by King [65], who shows how both the *switch* combinator of Yampa and the *switch* combinator of FrTime can be defined in terms of his *join* primitives.

One of the key aspects of FRP is *synchrony* (reactions are considered to be instantaneous). However, many of the concepts discussed in this thesis are also relevant in an *asynchronous* setting (where reactions can take a non-zero amount of time). A good example of this is *Fudgets* [16], an asynchronous reactive language embedded in Haskell. Fudgets was designed for programming graphical user interfaces, but it can also be used for other reactive domains. Like N-ary FRP (and UFRP), Fudgets is based around a first-class signal-function abstraction (called a *fudget*), and provides for feedback, dynamism and higher-order data-flow. The first-class status of fudgets is exploited for many of the benefits discussed in Section 3.5, including optimisation and the ability to freeze running fudgets. Similar issues also arise, such as the awkwardness of using combinators to express complex routing in a point-free style, and thus the need for some convenient syntax [15] (as discussed in Section 3.4.6). The key difference between Fudgets and FRP is that Fudgets has no concept of continuous-time signals, with the fudgets operating over discrete streams (analogous to FRP's event signals, except that, because of the asynchrony, occurrences are not fixed at specific time points). However, a fudget is not just a stream processor: unlike N-ary FRP (and UFRP), a fudget also has a connection to the outside world through which it can receive input and emit output.

10.2 Static Safety Checks

The synchronous data-flow languages [7, 46, 47] prevent undesirable network structures by performing static analyses at compile time. Domain-specific constraints such as causality [30] and initialisation of signals [21] can be checked in a fine-grained manner, but this often relies on the language having a static first-order structure. For example, the work on extending Lucid Synchrone with dynamism and higher-order data-flow [17, 22] has come at the cost of much more conservative analyses: explicit decoupling and initialisation of signals must appear syntactically within each node definition [104].

FRP approaches the problem from the other direction. Most FRP implementations are highly expressive, but lack totality and termination guarantees. In many cases this is unavoidable because the FRP variant is embedded in a host-language that lacks those guarantees. Nevertheless, some FRP implementations, similarly to the N-ary FRP embeddings described in this thesis, do guarantee that the *reactive* level of the language is total.

Real-Time FRP (RT-FRP) [128], a small and experimental CFRP variant, is one such language. The aim of RT-FRP was to establish time and space bounds on the reactive level, and thus totality was necessary. Infinite switching at a point in time is prevented by having residual signals start one-time-step after they are switched-in (time is modelled discretely). Note that this is distinct from decoupled switching combinators (Section 9.1.1), where the residual signal (or signal function) still *starts* at the moment of switching, even though the output is not observable until afterwards. Decoupled feedback is permitted in RT-FRP, but instantaneous feedback is disallowed by a specialised type system that only brings the signal identifier into scope in a recursive signal definition when it appears under a one-time-step–delay primitive. The exception to this is that the event signal that controls a switch may depend instantaneously on the overall value of the switching combinator, but this is safe because the residual signal function does not start until the next time step. Uninitialised signals are simply not definable. However, RT-FRP has very limited capabilities for abstracting over and combining reactive entities, essentially only being concerned with monolithic reactive expressions. There are no reactive constructs at the functional level: it is only used to perform pointwise operations on signals. A consequence of this is that RT-FRP is only structurally dynamic, not highly structurally dynamic; but this restriction is hard to avoid if space and time guarantees are required. In terms of implementation, the operational semantics of RT-FRP is similar to the decoupled transition functions of N-ary FRP with Feedback (Section 7.5.1), separating out the state update and computation of the signal value.

Version 1 of Elerea takes a somewhat different approach to avoiding ill-defined feedback, by having the implementation automatically insert one-time-step delays into instantaneous feedback loops [99]. This only applies to loops that contain a stateful signal function, so stateless feedback can still diverge. Nevertheless, this avoids some instances of deadlock, without placing restrictions on the programs that can be defined. However, this breaks referential transparency (as whether a delay is inserted into a stateful signal function depends upon the context in which that signal function is used), and the programmer does not know exactly where these delays are being inserted. Consequently, this feature has been depreciated from Version 2 of Elerea.

Recently, Krishnaswami and Benton [67] have studied non-expansive (causal) and contractive (decoupled) stream functions in the setting of FRP, with the aim of characterising precisely when feedback is well-defined. This is similar to the approach taken by N-ary FRP with Feedback (Chapter 7), though theirs is a discrete-time setting with first-class signals.

10.3 Optimisation of Reactive Languages

Incremental evaluation and change propagation have been studied extensively as optimisation techniques [3, 4, 108]. However, the problem becomes significantly more complex in a reactive setting. The notion of time passing leads to signal functions whose output can change even when their input does not, and structural dynamism means that many optimisation opportunities only arise at run-time. The former situation has been well-studied in the static first-order context of the synchronous data-flow languages [49, 70, 106], but the latter is a more open problem.

FrTime, a push-based FRP language embedded in the DrScheme environment [40], uses a variety of optimisation techniques [23]. The inherent change propagation of the push-based execution is enhanced by performing run-time equality checks on recomputed signal values to determine whether they really have changed [24]. It also uses a static optimisation called *lowering*, which reduces a data-flow network by fusing together composite signal functions into single signal functions (discarding the routing information) [14]. In FrTime, this technique is only applied to lifted pure functions. For example, (in the N-ary FRP setting) a typical lowering optimisation would be:

$(\mathit{lift}\ f \ggg \mathit{lift}\ g) \ \rightsquigarrow\ \mathit{lift}\ (g \circ f)$

FrTime's lowering optimisations are applied statically at compile time, which allows for substantial optimisation of source code, but does not allow dynamic optimisation of the network after structural switches. Lowering optimisations are also applied by Yampa [90] and Version 1 of Elerea [99], albeit not to the extent of FrTime. However, Yampa can lower some stateful signal functions as well as stateless ones. Yampa performs its lowering optimisations dynamically, which suffers from additional run-time overhead, but does allow for continued optimisation after structural switches. Experimentation in FrTime and Yampa has suggested that lowering is generally a worthwhile optimisation, but that in some cases it can have a negative impact.

This latter cases arises because of the implementation overhead in Yampa [90], and the loss of fine-grained change propagation in FrTime [23].

A recent development has been a static optimisation technique for *Causal Commutative Arrows* [75, 77] that lowers any static arrow network (which may include cycles) to a single arrow with a single internal state. When applied to some examples from Yampa (where a signal function is an arrow), the elimination of most of the arrow infrastructure has resulted in impressive[1] performance gains. However, this technique does not extend to networks containing switching combinators, and so cannot be applied to arbitrary Yampa programs (though it could be applied to static sub-networks).

Finally, recent work on Functional Hybrid Modelling has shown that, in certain cases, a just-in-time compilation technique can be a good fit for highly dynamic network structures [44]. At each structural switch the network is recompiled, allowing efficient execution of the (temporarily) static network between each structural switch.

10.4 Conclusions

Many FRP variants have semantic models. However, to my knowledge, the N-ary FRP model is the first to cater for multi-kinded signals in a setting with a first-class signal-function abstraction (where signals are second class).

Some reactive languages perform causality checks to ensure that ill-defined feedback cannot be formed, and some perform initialisation checks to ensure that undefined initial signal values are never used. There are also reactive languages that give totality guarantees for the reactive level of the language. However, such guarantees have not before been given in the presence of highly dynamic system structure.

Efficiently implementing FRP is the subject of ongoing research. Existing FRP implementations use a variety of approaches, and incorporate a wide range of optimisations. Many of the optimisations discussed in Chapter 8 of this thesis have proved beneficial in practice (such as in the latest version of Yampa [90]), but they have not yet been systematically applied in the way advocated here.

[1]Ranging between 5 to 200 times faster for the benchmarks tried [75].

Chapter 11

Summary and Future Work

This chapter summarises the content of this thesis, and discusses avenues for future work.

11.1 Summary

Chapter 1 introduced reactive programming, and the embedded approach to implementing domain-specific languages. Most implementations of FRP to date have been domain-specific embeddings within Haskell.

Chapter 2 introduced the notation of Agda, and the variant Agda syntax used to express the semantics and example code in this thesis.

Chapter 3 motivated the functional approach to reactive programming (Section 3.1); introduced the fundamental concepts of FRP (Section 3.2); and described CFRP (Section 3.3) and UFRP (Section 3.4), two conceptual models on which several FRP variants are based. The distinguishing features between the two are that CFRP has first-class signals (and signal generators) and distinct discrete-time and continuous-time signals, whereas UFRP is based around a first-class signal-function abstraction and only really has one signal kind. The pros and cons of each approach were then discussed, both in terms of semantics and implementation (Section 3.5). Two points of particular note are that *pure* implementations of first-class signals have led to space and time leaks, and that the UFRP model hides much of the network routing thus limiting the scope for optimisation.

Chapter 4 introduced a new FRP language called N-ary FRP. While this language was inspired by UFRP, it has a number of important differences including *three distinct signal kinds* and *n-ary signal functions*. These features were introduced to allow kind-specific operations on signals while overcoming the routing limitations of UFRP. To my knowledge, this is the first FRP language to combine multi-kinded signals with a first-class signal-function abstraction.

Chapter 5 described an embedded implementation of N-ary FRP in both Agda and Haskell. The implementation is essentially the same in both languages, the differences being due to the differing type systems of the two host languages. These embeddings serve three purposes. First, they demonstrate that signal-function–based FRP can be implemented purely and simply. Second, the Haskell embedding confirms that the N-ary FRP type system is suitable for embedding in a mainstream functional language. Third, the Agda embedding guarantees the totality of the implementation and of the example N-ary FRP programs. While this does not *prove* anything about the corresponding Haskell embedding, the close correspondence between the two gives a high level of confidence.

Chapter 6 introduced temporal logic, and used it to express several temporal properties of signals and signal functions in terms of the N-ary FRP conceptual model. In particular, the properties of *causality*, *decoupledness* and *strict decoupledness* were formalised.

Chapter 7 extended N-ary FRP with a reactive-level feedback combinator. Through a type-system refinement, the decoupledness and strict decoupledness properties were used to justify the claim that all feedback expressible by this combinator is well-defined. To my knowledge, this is the first continuous-time semantics of FRP that allows feedback while guaranteeing that it is well-defined.

Chapter 8 considered the notion of change in the setting of N-ary FRP, and the change-based optimisations that are possible. First the push- and pull-based approaches to implementing FRP were discussed (Section 8.1). A signal-kind–specific notion of change was then defined, along with a number of change-based properties of signals and signal functions (Section 8.3). Finally, two approaches to change-based optimisation were considered: *structural optimisation* and *change propagation* (Section 8.5). These optimisations have not yet been applied to an N-ary FRP implementation, but, as discussed, many of them appear in a variety of forms in other FRP implementations.

Chapter 9 considered extensions to N-ary FRP. Section 9.1 discussed the problems that arise when attempting to extend the N-ary FRP model to account for event occurrences (and changes in Step signals) *immediately after* a time point. Despite the conceptual challenges, such an extension seems necessary to account for several useful primitives that appear in many FRP implementations. Section 9.2 extended N-ary FRP with *uninitialised signals*. A type-system refinement ensures that the initial undefined value of such a signal is never used. Guaranteeing the correct use of uninitialised signals has been studied before [21], but, to my knowledge, not in a highly structurally dynamic setting, nor in the presence of distinct signal kinds. Section 9.3 further refined the type system of N-ary FRP with Feedback by using Boolean matrices to precisely track instantaneous dependencies between signals.

Finally, Chapter 10 discussed related work (omitting that which had already been described in the individual chapters).

All of the code in this thesis has been formulated in Agda without the syntactic sugar used for presentational purposes (see Section 2.2). However, not all of the lemmas stated have yet been formally proved (the proofs completed thus far are available in the online archive [1]). Specifically, the properties of the dynamic combinators and the more complicated primitive signal functions remain to be verified. This is not due to any particular technical difficulty, but rather that proving properties in Agda is a time-consuming and tedious process. That said, formal verification in Agda has been extremely helpful while developing the N-ary FRP model and temporal properties, particularly in identifying counter-intuitive aspects of the model.

11.2 Future Work

Yampa provides a set of collection-based switching combinators that allow dynamic collections of signal functions to be maintained [92]. Signal functions can be added to and removed from such collections during execution. This capability has proved extremely useful in applications such as video games [27], visual tracking [92], and sound synthesis [43]. It would be interesting to extend N-ary FRP with such combinators, and to determine if they can be encoded in terms of *switch* and *freeze*, or whether additional primitive dynamic combinators are required.

The combinators used for structuring N-ary FRP are based on the Arrow Framework [59]. Programming directly with such combinators is awkward for more complicated arrows, and so a syntactic sugar was devised to aid writing of arrow code [101]. That syntax cannot be leveraged directly for N-ary FRP, but something similar (such as that in Section 7.4.2) would certainly be a desirable feature in an N-ary FRP implementation. For example, the FHM language *Hydra* [44] uses an arrow-inspired syntax for expressing signal relations, with a recent implementation using Haskell's quasiquoting to interpret the syntax [42]. A similar approach could be taken by an N-ary FRP implementation. Another recent development has been the *Arrow Calculus* [74], an alternative (and equivalent) notation for arrows, structured more along the lines of traditional lambda calculus. The Arrow Calculus does not yet support feedback, but a syntax based on the Arrow Calculus for programming at the reactive level seems appealing (replacing the five routing primitives).

Reconfigurable systems, including those that receive software updates whilst running, are becoming increasingly prevalent [22, 53]. Examples can be found in common items such as digital televisions and mobile telephones, as well as in safety-critical systems such as air-traffic control. Semantically, FRP's capacity for dynamism and higher-order data-flow is ideal to express such systems, as new programs (signal functions) can be received as system inputs. This is not yet possible in any FRP *implementations*, but there has been work in Haskell to allow dynamic loading of new code [98, 119], which would make a good starting point for future work in this direction.

The optimisations discussed in Chapter 8 have yet to be applied to a realistic implementation of N-ary FRP. While many of the optimisations appear in other languages in various forms, they need to be tested in a systematic way to determine their effectiveness. More generally, there is a lack of a standard set of benchmarks for FRP. Such a set would allow the expressiveness, efficiency and optimisation techniques of the different varieties of FRP to be compared more thoroughly than is currently the case.

Many of the domain-specific constraints described in this thesis, such as ensuring the absence of instantaneous feedback (Chapter 7) and ensuring that the initial value of an uninitialised signal is never used (Section 9.2), have been encoded using type-level Booleans. It would be interesting to embed N-ary FRP in a language with an in-built Boolean constraint solver, such as Dependent ML [130], which should avoid the need for the manual resolving of trivial Boolean constraints that is required in the embeddings described in this thesis.

As discussed in Section 9.1, a natural extension of N-ary FRP is to consider events and changes that happen *immediately after* a time point. Such functionality is already present in many discretely sampled FRP implementations, where *immediately after* is approximated by the next time step. However the N-ary FRP model cannot express such functionality. The most promising approach for extending N-ary FRP in this direction would seem to be one based on *super-dense time* [81, 84, 132].

Finally, note that the only implementations of N-ary FRP thus far are the proof-of-concept prototypes described in this thesis (chapters 5 and 7). The long-term aim of this work is an efficient scalable implementation of N-ary FRP that respects the conceptual model and incorporates domain-specific safety constraints.

Bibliography

[1] URL `www.cs.nott.ac.uk/Research/fop/nas-thesis-code.tar.gz`.

[2] *Simulink User's Guide, Version 7.6*. 3 Apple Hill Drive, Natick, MA, 2010. URL `www.mathworks.com/help/toolbox/simulink/`.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Principles of Programming Languages (POPL '02)*, pages 247–259. ACM, 2002.

[4] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Principles of Programming Languages (POPL '08)*, pages 309–322. ACM, 2008.

[5] Stefan Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3:133–181, 1922.

[6] Albert Benveniste and Gérard Berry. Real-time systems design and programming. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[7] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE, Special issue on embedded systems*, 91(1):64–83, 2003.

[8] Gérard Berry. Real time programming: Special purpose or general purpose languages. In *IFIP Congress*, pages 11–17, 1989.

[9] Gérard Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[10] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[11] Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

[12] Kristopher J. Blom. *Dynamic Interactive Virtual Environments*. PhD thesis, Department of Informatics, University of Hamburg, 2009.

[13] Ana Bove and Peter Dybjer. Dependent types at work. In *Language Engineering and Rigorous Software Development International LerNet ALFA Summer School*, pages 57–99. Springer, 2008.

[14] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A static optimization technique for transparent functional reactivity. In *Partial Evaluation and Program Manipulation (PEPM '07)*, pages 71–80. ACM, 2007.

[15] Magnus Carlsson. ProdArrows — arrows for fudgets, 2001. URL `www.carlssonia.org/~magnus/ogi/ProdArrows`.

[16] Magnus Carlsson and Thomas Hallgren. *Fudgets — Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, 1998.

[17] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming (ICFP '96)*, pages 226–238. ACM, 1996.

[18] Adam Cataldo, Edward Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. Discrete-event systems: Generalizing metric spaces and fixed-point semantics. Technical report, EECS Department, University of California, Berkeley, 2005.

[19] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, 2006.

[20] Mun Hon Cheong. Functional programming and 3D games. BEng thesis, University of New South Wales, 2005.

[21] Jean-Louis Colaço and Marc Pouzet. Type-based initialization analysis of a synchronous data-flow language. *Software Tools for Technology Transfer*, 6(3):245–255, 2004.

[22] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *Embedded Software (EMSOFT '04)*, pages 230–239. ACM, 2004.

[23] Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Department of Computer Science, Brown University, 2008.

[24] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming (ESOP '06)*, pages 294–308. Springer, 2006.

[25] Antony Courtney. *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University, 2004.

[26] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop (Haskell '01)*, pages 41–69. Elsevier, 2001.

[27] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Haskell Workshop (Haskell '03)*, pages 7–18. ACM, 2003.

[28] Duncan Coutts. Partial evaluation for domain-specific embedded languages in a higher order typed language. Transfer dissertation, Oxford University, 2004.

[29] P.J.L. Cuijpers, M.A. Reniers, and A.G. Engels. Beyond Zeno-behaviour. Technical report, Department of Computer Science, Eindhoven University of Technology, 2001.

[30] Pascal Cuoq and Marc Pouzet. Modular causality in a synchronous stream language. In *European Symposium on Programming (ESOP '01)*, pages 237–251. Springer, 2001.

[31] Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[32] Anthony Daniels. *A Semantics for Functions and Behaviours.* PhD thesis, University of Nottingham, 1999.

[33] Nils Anders Danielsson. Beating the productivity checker using embedded languages. In *Partiality and Recursion in Interactive Theorem Provers (PAR '10)*, pages 29–48. EPTCS, 2010.

[34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating System Design and Implementation (OSDI '04)*, pages 137–150. USENIX Association, 2004.

[35] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, 2003.

[36] Conal Elliott. Functional implementations of continuous modeled animation. In *Programming Language Implementation and Logic Programming / Algebraic and Logic Programming (PLILP/ALP '98)*, pages 284–299. Springer, 1998.

[37] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium (Haskell '09)*, pages 25–36. ACM, 2009.

[38] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP '97)*, pages 263–273. ACM, 1997.

[39] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[40] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[41] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI '91)*, pages 268–277. ACM, 1991.

[42] George Giorgidze and Henrik Nilsson. Embedding a functional hybrid modelling language in Haskell. In *Implementation and Application of Functional Languages (IFL '08).* To appear.

[43] George Giorgidze and Henrik Nilsson. Switched-on Yampa: Declarative programming of modular synthesizers. In *Practical Aspects of Declarative Languages (PADL '08)*, pages 282–298. Springer, 2008.

[44] George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Functional and Constraint Logic Programming (WFLP '10)*, pages 48–65. Springer, 2011.

[45] Louis-Julien Guillemette and Stefan Monnier. One vote for type families in Haskell! In *Trends in Functional Programming (TFP '08)*, pages 81–96. Intellect, 2009.

[46] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. The Springer International Series in Engineering and Computer Science. Springer, 1993.

[47] Nicolas Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Computer Aided Verification (CAV '98)*, pages 1–16. Springer, 1998.

[48] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[49] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *Programming Language Implementation and Logic Programming (PLILP '91)*, pages 207–218. Springer, 1991.

[50] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498. 1985.

[51] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *International Conference on Functional Programming (ICFP '03)*, pages 3–13. ACM, 2003.

[52] Thomas A. Henzinger. The theory of hybrid automata. In *Logics in Computer Science (LICS '96)*, pages 278–292. IEEE Computer Society, 1996.

[53] Michael Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.

[54] William A. Howard. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[55] Paul Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse (ICSR '98)*, pages 134–142. IEEE Computer Society, 1998.

[56] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[57] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming (AFP '02)*, pages 159–187. Springer, 2003.

[58] John Hughes. Why functional programming matters. In *Research Topics in Functional Programming*, pages 17–42. Addison-Wesley, 1990.

[59] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37 (1–3):67–111, 2000.

[60] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[61] Graham Hutton and Mauro Jaskelioff. Representing contractive functions on streams. In preparation, 2011.

[62] Wolfgang Jeltsch. Improving push-based FRP. In *Draft Proceedings of Trends in Functional Programming (TFP '08)*, pages 179–193, 2008.

[63] Wolfgang Jeltsch. Signals, not generators! In *Trends in Functional Programming (TFP '09)*, pages 145–160. Intellect, 2010.

[64] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP '06)*, pages 50–61. ACM, 2006.

[65] Christopher T. King. An axiomatic semantics for functional reactive programming. Master's thesis, Worcester Polytechnic Institute, 2008.

[66] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In *Reflections on the Work of C.A.R. Hoare*, chapter 14, pages 301–331. Springer, 2010.

[67] Neelakantan R. Krishnaswami and Nick Benton. An ultrametric model of reactive programming. Unpublished, 2010. URL `http://research.microsoft.com/apps/pubs/?id=135433`.

[68] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[69] Edward A. Lee. Embedded software. *Advances in Computers*, 56:56–97, 2002.

[70] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[71] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Embedded Software (EMSOFT '07)*, pages 114–123. ACM, 2007.

[72] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Domain-Specific Languages (DSL '99)*, pages 109–122. ACM, 1999.

[73] Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical report, School of Computer Science, Carnegie Mellon University, 2005.

[74] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. Technical report, School of Informatics, University of Edinburgh, 2008.

[75] Hai Liu. *The Theory and Practice of Causal Commutative Arrows*. PhD thesis, Department of Computer Science, Yale University, 2011.

[76] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, 2007.

[77] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *International Conference on Functional Programming (ICFP '09)*, pages 35–46. ACM, 2009.

[78] Roberto Lublinerman and Stavros Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe (DATE '08)*, pages 1504–1509, 2008.

[79] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Principles of Programming Languages (POPL '09)*, pages 78–89. ACM, 2009.

[80] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *International Conference on Functional Programming (ICFP '08)*, pages 335–345. ACM, 2008.

[81] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice*, pages 447–484. Springer, 1992.

[82] Sharad Malik. Analysis of cyclic combinational circuits. *Computer-Aided Design*, 13(7): 950–956, 1994.

[83] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[84] Zohar Manna and Amir Pnueli. Verifying hybrid systems. *Hybrid Systems*, 736:4–35, 1993.

[85] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[86] Simon Marlow. *The Haskell Report*, 2010. URL `www.haskell.org`.

[87] William I. McLaughlin. Thomson's Lamp is dysfunctional. *Synthese*, 116(3):281–301, 1998.

[88] Stefan Monnier and David Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Programming Languages meets Program Verification (PLPV '10)*, pages 1–8. ACM, 2010.

[89] Henrik Nilsson. Functional automatic differentiation with Dirac impulses. In *International Conference on Functional Programming (ICFP '03)*, pages 153–164. ACM, 2003.

[90] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *International Conference on Functional Programming (ICFP '05)*, pages 54–65. ACM, 2005.

[91] Henrik Nilsson. Type-based structural analysis for modular systems of equations. *Simulation News Europe*, 19(1):17–28, 2009.

[92] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop (Haskell '02)*, pages 51–64. ACM, 2002.

[93] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Practical Aspects of Declarative Languages (PADL '03)*, pages 376–390. Springer, 2003.

[94] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[95] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[96] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming (AFP '08)*, pages 230–266. Springer, 2009.

[97] Clemens Oertel. *RatTracker: A Functional-Reactive Approach to Flexible Control of Behavioural Conditioning Experiments*. PhD thesis, Wilhelm-Schickard-Institute for Computer Science, University of Tübingen, 2006.

[98] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell in. In *Haskell Workshop (Haskell '04)*, pages 10–21. ACM, 2004.

[99] Gergely Patai. Eventless reactivity from scratch. In *Draft Proceedings of Implementation and Application of Functional Languages (IFL '09)*, pages 126–140, 2009.

[100] Gergely Patai. Efficient and compositional higher-order streams. In *Functional and Constraint Logic Programming (WFLP '10)*. Springer, 2011.

[101] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming (ICFP '01)*, pages 229–240. ACM, 2001.

[102] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages (PADL '99)*, pages 91–105. Springer, 1999.

[103] John Peterson, Paul Hudak, Alastair Reid, and Greg Hager. FVision: A declarative language for visual tracking. In *Practical Aspects of Declarative Languages (PADL '01)*, pages 304–321. Springer, 2001.

[104] Marc Pouzet. *Lucid Synchrone, version 3: Tutorial and reference manual*. Université Paris-Sud, LRI, 2006. URL `www.di.ens.fr/~pouzet/lucid-synchrone`.

[105] Marc Pouzet. On combining synchronous and functional programming. Presentation at Between Control and Software: Workshop in honor of Paul Caspi, 2007. URL `www.artist-embedded.org/artist/Programme,1140.html`.

[106] Marc Pouzet and Pascal Raymond. Modular static scheduling of synchronous data-flow networks: An efficient symbolic representation. *Design Automation for Embedded Systems*, 14(3):165–192, 2010.

[107] Arthur N. Prior. *Past, Present and Future*. Oxford University Press, 1967.

[108] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages (POPL '93)*, pages 502–510. ACM, 1993.

[109] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *International Conference on Functional Programming (ICFP '08)*, pages 51–62. ACM, 2008.

[110] Neil Sculthorpe and Henrik Nilsson. Keeping calm in the face of change: Towards optimisation of FRP by reasoning about change. *Higher-Order and Symbolic Computation*. To appear.

[111] Neil Sculthorpe and Henrik Nilsson. Optimisation of dynamic, hybrid signal function networks. In *Trends in Functional Programming (TFP '08)*, pages 97–112. Intellect, 2009.

[112] Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. In *International Conference on Functional Programming (ICFP '09)*, pages 23–34. ACM, 2009.

[113] Sean Seefried, Manuel M. T. Chakravarty, and Gabriele Keller. Optimising embedded DSLs using Template Haskell. In *Generative Programming and Component Engineering (GPCE '04)*, pages 186–205. Springer, 2004.

[114] Tim Sheard. Type-level computation using narrowing in Ωmega. In *Programming Languages meets Program Verification (PLPV '06)*, pages 105–128, 2006.

[115] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop (Haskell '02)*, pages 1–16. ACM, 2002.

[116] Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kinds = dependent programming. Technical report, Portland State University, 2005.

[117] Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference*, pages 328–333. IEEE Computer Society, 1996.

[118] Ben A. Sijtsma. On the productivity of recursive list definitions. *Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[119] Don Stewart. *Dynamic Extension of Typed Functional Languages.* PhD thesis, School of Computer Science and Engineering, University of New South Wales, 2010.

[120] Jean-Pierre Talpin and David Nowak. A synchronous semantics of higher-order processes for modeling reconfigurable reactive systems. In *Foundations of Software Technology and Theoretical Computer Science (FST & TCS '98)*, pages 78–89. Springer, 1998.

[121] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.0.3*, 2011. URL `www.haskell.org/ghc`.

[122] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, second edition, 1999.

[123] James F. Thomson. Tasks and super-tasks. *Analysis*, 15(1):1–13, 1954.

[124] Yde Venema. Temporal logic. In *The Blackwell Guide to Philosophical Logic*, chapter 10, pages 203–223. Blackwell, 2001.

[125] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Suzmann. Modular type inference with local assumptions. *Journal of Functional Programming.* To appear.

[126] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press Professional, 1985.

[127] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Programming Language Design and Implementation (PLDI '00)*, pages 242–252. ACM, 2000.

[128] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, pages 146–156. ACM, 2001.

[129] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages (PADL '02)*, pages 155–172. Springer, 2002.

[130] Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

[131] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Principles of Programming Languages (POPL '99)*, pages 214–227. ACM, 1999.

[132] Haiyang Zheng. *Operational Semantics of Hybrid Systems.* PhD thesis, EECS Department, University of California, Berkeley, 2007.

Appendix A

Utility Functions

This appendix contains the utility functions used in this thesis that are not defined elsewhere. These functions are defined in the syntax of AgdaFRP (Section 2.2), but in most cases translating them into Haskell and Agda is trivial. Consequently, these functions are also used in the Haskell and Agda embeddings without repeating their definitions in Haskell and Agda syntax. A few of these functions cannot easily be translated into Haskell, but those functions are not used in the Haskell embedding.

Ideal real numbers, along with standard arithmetic and comparative operations on them, are assumed rather than defined. In the Agda encoding of this thesis, real numbers and the operations on them are postulated as axioms.

A.1 Combinators

First are several function combinators:

```
_∘_ : (B → C) → (A → B) → (A → C)
g ∘ f = λ a → g (f a)

const : A → B → A
const a _ = a

flip : (A → B → C) → B → A → C
flip f b a = f a b

result : (B → C) → (A → B) → (A → C)
result f g = f ∘ g

result2 : (C → D) → (A → B → C) → (A → B → D)
result2 f g a = f ∘ g a
```

Next are combinators over product types (many of which have lifted counterparts at the reactive level of N-ary FRP):

```
curry : (A × B → C) → (A → B → C)
curry f a b = f (a, b)

uncurry : (A → B → C) → (A × B → C)
uncurry f (a, b) = f a b

fork : A → A × A
fork a = (a, a)

swap : A × B → B × A
swap (a, b) = (b, a)
```

$$
\begin{aligned}
&\mathit{fst} \;:\; A \times B \to A\\
&\mathit{fst}\ (a, b) \;=\; a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{snd} \;:\; A \times B \to B\\
&\mathit{snd}\ (a, b) \;=\; b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{first} \;:\; (A \to C) \to A \times B \to C \times B\\
&\mathit{first}\ f\ (a, b) \;=\; (f\ a, b)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{second} \;:\; (B \to C) \to A \times B \to A \times C\\
&\mathit{second}\ f\ (a, b) \;=\; (a, f\ b)
\end{aligned}
$$

A.2 Booleans

Logical negation is defined as follows:

$$
\begin{aligned}
&\mathit{not} \;:\; \mathit{Bool} \to \mathit{Bool}\\
&\mathit{not}\ \mathsf{false} \;=\; \mathsf{true}\\
&\mathit{not}\ \mathsf{true} \;=\; \mathsf{false}
\end{aligned}
$$

A Boolean can be converted into a proposition:

$$
\begin{aligned}
&\mathit{isTrue} \;:\; \mathit{Bool} \to \mathit{Set}\\
&\mathit{isTrue}\ \mathsf{true} \;=\; \mathit{True}\\
&\mathit{isTrue}\ \mathsf{false} \;=\; \mathit{False}
\end{aligned}
$$

A.3 Lists

The following list functions are standard:

$$
\begin{aligned}
&\mathit{map} \;:\; (A \to B) \to \mathit{List}\ A \to \mathit{List}\ B\\
&\mathit{map}\ f\ [\,] &&= [\,]\\
&\mathit{map}\ f\ (a :: \mathit{as}) &&= f\ a :: \mathit{map}\ f\ \mathit{as}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reverse} \;:\; \mathit{List}\ A \to \mathit{List}\ A\\
&\mathit{reverse}\ [\,] &&= [\,]\\
&\mathit{reverse}\ (a :: \mathit{as}) &&= \mathit{reverse}\ \mathit{as} +\!\!+ (a :: [\,])
\end{aligned}
$$

$$
\begin{aligned}
&\_ +\!\!+ \_ \;:\; \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A\\
&[\,] +\!\!+ \mathit{bs} &&= \mathit{bs}\\
&(a :: \mathit{as}) +\!\!+ \mathit{bs} &&= a :: (\mathit{as} +\!\!+ \mathit{bs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{zipWith} \;:\; (A \to B \to C) \to \mathit{List}\ A \to \mathit{List}\ B \to \mathit{List}\ C\\
&\mathit{zipWith}\ f\ (a :: \mathit{as})\ (b :: \mathit{bs}) &&= f\ a\ b :: \mathit{zipWith}\ f\ \mathit{as}\ \mathit{bs}\\
&\mathit{zipWith}\ f\ \_\ \_ &&= [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dropWhile} \;:\; (A \to \mathit{Bool}) \to \mathit{List}\ A \to \mathit{List}\ A\\
&\mathit{dropWhile}\ p\ [\,] &&= [\,]\\
&\mathit{dropWhile}\ p\ (a :: \mathit{as}) &&= \mathbf{if}\ p\ a\ \mathbf{then}\ \mathit{dropWhile}\ p\ \mathit{as}\ \mathbf{else}\ a :: \mathit{as}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sum} \;:\; \mathit{List}\ \mathbb{R} \to \mathbb{R}\\
&\mathit{sum}\ [\,] &&= 0\\
&\mathit{sum}\ (x :: \mathit{xs}) &&= x + \mathit{sum}\ \mathit{xs}
\end{aligned}
$$

A.4 Maybe

Several basic combinators over options types are required:

$$
\begin{aligned}
&\mathit{fromMaybe} \;:\; A \to \mathit{Maybe}\ A \to A\\
&\mathit{fromMaybe}\ a\ \mathsf{nothing} &&= a\\
&\mathit{fromMaybe}\ a\ (\mathsf{just}\ b) &&= b
\end{aligned}
$$

$maybeMap\ :\ (A \to B) \to Maybe\ A \to Maybe\ B$
$maybeMap\ f\ \mathsf{nothing}\ =\ \mathsf{nothing}$
$maybeMap\ f\ (\mathsf{just}\ a)\ =\ \mathsf{just}\ (f\ a)$

$maybeMap2\ :\ (A \to B \to C) \to Maybe\ A \to Maybe\ B \to Maybe\ C$
$maybeMap2\ f\ \mathsf{nothing}\ mb\ =\ \mathsf{nothing}$
$maybeMap2\ f\ (\mathsf{just}\ a)\ mb\ =\ maybeMap\ (f\ a)\ mb$

$maybeMerge\ :\ (A \to C) \to (B \to C) \to (A \to B \to C) \to Maybe\ A \to Maybe\ B \to Maybe\ C$
$maybeMerge\ f_a\ f_b\ f_{ab}\ \mathsf{nothing}\ \mathsf{nothing}\ =\ \mathsf{nothing}$
$maybeMerge\ f_a\ f_b\ f_{ab}\ \mathsf{nothing}\ (\mathsf{just}\ b)\ =\ \mathsf{just}\ (f_b\ b)$
$maybeMerge\ f_a\ f_b\ f_{ab}\ (\mathsf{just}\ a)\ \mathsf{nothing}\ =\ \mathsf{just}\ (f_a\ a)$
$maybeMerge\ f_a\ f_b\ f_{ab}\ (\mathsf{just}\ a)\ (\mathsf{just}\ b)\ =\ \mathsf{just}\ (f_{ab}\ a\ b)$

$maybeFilter\ :\ (A \to Bool) \to Maybe\ A \to Maybe\ A$
$maybeFilter\ p\ \mathsf{nothing}\ =\ \mathsf{nothing}$
$maybeFilter\ p\ (\mathsf{just}\ a)\ =\ \mathbf{if}\ p\ a\ \mathbf{then}\ \mathsf{just}\ a\ \mathbf{else}\ \mathsf{nothing}$

A.5 Intervals

A data type for time intervals is defined as follows:

$\mathbf{data}\ Interval\ :\ Set\ \mathbf{where}$
$\quad \langle\ \_,\_\ \rangle\ :\ Time \to Time \to Interval$
$\quad \langle\ \_,\_\ ]\ :\ Time \to Time \to Interval$
$\quad [\ \_,\_\ \rangle\ :\ Time \to Time \to Interval$
$\quad [\ \_,\_\ ]\ :\ Time \to Time \to Interval$

Note that parentheses are reserved syntax in Agda, so angled brackets are used to denote open intervals.

A proposition that expresses a time value being within an interval is defined as follows:

$\_\in\_\ :\ Time \to Interval \to Set$
$t \in \langle\ t_1, t_2\ \rangle\ =\ (t_1 < t) \times (t < t_2)$
$t \in \langle\ t_1, t_2\ ]\ =\ (t_1 < t) \times (t \leqslant t_2)$
$t \in [\ t_1, t_2\ \rangle\ =\ (t_1 \leqslant t) \times (t < t_2)$
$t \in [\ t_1, t_2\ ]\ =\ (t_1 \leqslant t) \times (t \leqslant t_2)$

Finally, $ConstantOver\ s\ i$ holds if the time-varying value s is constant over the interval i:

$ConstantOver\ :\ (Time \to A) \to Interval \to Set$
$ConstantOver\ s\ i\ =\ (t_1\ t_2\ :\ Time) \to t_1 \in i \to t_2 \in i \to s\ t_1 \equiv s\ t_2$

Appendix B

N-ary FRP Conceptual Definitions

This appendix contains utility functions that operate on the conceptual model of N-ary FRP (defined in Section 4.1). Additionally, many of the N-ary FRP primitives in Section 4.2 were introduced without defining their semantics: this appendix contains those definitions. These definitions make use of the utility functions in Appendix A.

B.1 Utility Functions

First are some functions on change lists and change prefixes. The time of the last change in a change list is computed by summing its time deltas:

```
lastChangeTime : ChangeList A → Time
lastChangeTime = sum ∘ map fst
```

The prefix of a change list up to a time point (inclusive or exclusive) is given by:

```
takeIncl : Time → ChangeList A → ChangeList A
takeIncl _ []                        = []
takeIncl t ((δ, a) :: δas)  | t < δ  = []
                            | t ⩾ δ  = (δ, a) :: takeIncl (t − δ) δas

takeExcl : Time → ChangeList A → ChangeList A
takeExcl _ []                        = []
takeExcl t ((δ, a) :: δas)  | t ⩽ δ  = []
                            | t > δ  = (δ, a) :: takeExcl (t − δ) δas
```

Whether there is a change at a time point in a change list or change prefix is given by:

```
lookupCL : ChangeList A → Time → Maybe A
lookupCL []              _          = nothing
lookupCL ((δ, a) :: δas) t | t < δ  = nothing
                           | t ≡ δ  = just a
                           | t > δ  = lookupCL δas (t − δ)

lookupCP : ChangePrefix A → Time → Maybe A
lookupCP cp t = lookupCL (cp t) t
```

Next are some functions over signals. The value of a Step signal at a time point is given by:

```
val : SigVec (S A) → Time → A
val (a₀, cp) t = case reverse (cp t) of
                   []             → a₀
                   (_, a₁) :: _   → a₁
```

The value of a Step signal *immediately prior* to a time point (its *left limit*) is given by:

```
leftLimit : SigVec (S A) → Time⁺ → A
leftLimit (a₀, cp) t = case reverse (takeExcl t (cp t)) of
                         []             → a₀
                         (_, a₁) :: _   → a₁
```

Whether an event is occurring at a time point, and its value if so, is given by:

```
occ : SigVec (E A) → Time → Maybe A
occ (ma, cp) t | t ≡ 0 = ma
               | t > 0 = lookupCP cp t
```

Finally, the first event occurrence in an Event signal, provided it occurs before a specified time point (inclusive), is given by:

```
fstOcc : SigVec (E A) → Time → Maybe (Time × A)
fstOcc (just a, _)     _ = just (0, a)
fstOcc (nothing, cp) t   = case cp t of
                             []        → nothing
                             δa :: _   → just δa
```

B.2 Lifting Functions

The lifting functions take a pure function from the host language and apply it pointwise to a signal. Before defining those primitives, some pointwise mappings over change lists, change prefixes and signals are defined as follows:

```
mapCL : (A → B) → ChangeList A → ChangeList B
mapCL = map ∘ second

mapCP : (A → B) → ChangePrefix A → ChangePrefix B
mapCP = result ∘ mapCL

mapC : (A → B) → SigVec (C A) → SigVec (C B)
mapC = result

mapE : (A → B) → SigVec (E A) → SigVec (E B)
mapE f (ma, cp) = (maybeMap f ma, mapCP f cp)

mapS : (A → B) → SigVec (S A) → SigVec (S B)
mapS f (a, cp) = (f a, mapCP f cp)
```

Sometimes it is necessary to map over two signals. In the case of Continuous signals, this is straightforward:

```
mapC2 : (A → B → Z) → SigVec (C A) → SigVec (C B) → SigVec (C Z)
mapC2 f s₁ s₂ t = f (s₁ t) (s₂ t)
```

However, for Step and Event signals this is more complicated. When mapping over two Step signals, the resultant signal should contain a change whenever there is a change in either argument signal. Furthermore, that change needs to reflect the most recent value of the other signal, even if that other signal has not changed at the same time (as will usually be the case). This is defined as follows:

```
mapS2 : (A → B → Z) → SigVec (S A) → SigVec (S B) → SigVec (S Z)
mapS2 f (a, cp_a) (b, cp_b) = (f a b, λ t → mergeS a b (cp_a t) (cp_b t))
  where
    mergeS : A → B → ChangeList A → ChangeList B → ChangeList Z
    mergeS a_0 b_0 [] δbs = mapCL (f a_0) δbs
    mergeS a_0 b_0 δas [] = mapCL (flip f b_0) δas
    mergeS a_0 b_0 ((δ_a, a_1) :: δas) ((δ_b, b_1) :: δbs)
      | δ_a ≡ δ_b = (δ_a, f a_1 b_1) :: mergeS a_1 b_1 δas δbs
      | δ_a < δ_b = (δ_a, f a_1 b_0) :: mergeS a_1 b_0 δas ((δ_b − δ_a, b_1) :: δbs)
      | δ_a > δ_b = (δ_b, f a_0 b_1) :: mergeS a_0 b_1 ((δ_a − δ_b, a_1) :: δas) δbs
```

As discussed in Section 4.2.3, there are two ways to map over two Event signals: *merging* (keeping occurrences from both signals), or *joining* (only keeping temporally intersecting occurrences):

```
mergeE2 : (A → Z) → (B → Z) → (A → B → Z) → SigVec (E A) → SigVec (E B) → SigVec (E Z)
mergeE2 f_a f_b f_ab (ma, cp_a) (mb, cp_b) = (maybeMerge f_a f_b f_ab ma mb, λ t → mergeCL (cp_a t) (cp_b t))
  where
    mergeCL : ChangeList A → ChangeList B → ChangeList Z
    mergeCL [] δbs = mapCL f_b δbs
    mergeCL δas [] = mapCL f_a δas
    mergeCL ((δ_a, a) :: δas) ((δ_b, b) :: δbs) | δ_a ≡ δ_b = (δ_a, f_ab a b) :: mergeCL δas δbs
                                                | δ_a < δ_b = (δ_a, f_a a) :: mergeCL δas (((δ_b − δ_a), b) :: δbs)
                                                | δ_a > δ_b = (δ_b, f_b b) :: mergeCL (((δ_a − δ_b), a) :: δas) δbs
```

```
joinE2 : (A → B → Z) → SigVec (E A) → SigVec (E B) → SigVec (E Z)
joinE2 f (ma, cp_a) (mb, cp_b) = (maybeMap2 f ma mb, λ t → joinCL 0 (cp_a t) (cp_b t))
  where
    joinCL : Time → ChangeList A → ChangeList B → ChangeList Z
    joinCL _ [] _ = []
    joinCL _ _ [] = []
    joinCL d ((δ_a, a) :: δas) ((δ_b, b) :: δbs) | δ_a ≡ δ_b = (d + δ_a, f a b) :: joinCL 0 δas δbs
                                                 | δ_a < δ_b = joinCL (d + δ_a) δas ((δ_b − δ_a, b) :: δbs)
                                                 | δ_a > δ_b = joinCL (d + δ_b) ((δ_a − δ_b, a) :: δas) δbs
```

The *sampleWith* lifting functions map over two signals of different kinds: a Continuous or Step signal, and an Event signal. Corresponding mapping functions are defined as follows:

```
mapCE : (A → B → Z) → SigVec (C A) → SigVec (E B) → SigVec (E Z)
mapCE f s (mb, cp) = (maybeMap (f (s 0)) mb, λ t → mergeCE 0 (cp t))
  where
    mergeCE : Time → ChangeList B → ChangeList Z
    mergeCE d []             = []
    mergeCE d ((δ, b) :: δbs) = let d′ = d + δ in (d′, f (s d′) b) :: mergeCE d′ δbs
```

```
mapSE : (A → B → Z) → SigVec (S A) → SigVec (E B) → SigVec (E Z)
mapSE f s (mb, cp) = (maybeMap (f (val s 0)) mb, λ t → mergeSE 0 (cp t))
  where
    mergeSE : Time → ChangeList B → ChangeList Z
    mergeSE d []             = []
    mergeSE d ((δ, b) :: δbs) = let d′ = d + δ in (d′, f (val s d′) b) :: mergeSE d′ δbs
```

Using these mappings, defining the semantics of the lifting functions is trivial:

```
liftC : (A → B) → SF (C A) (C B)
liftC f ≈ mapC f

liftS : (A → B) → SF (S A) (S B)
liftS f ≈ mapS f

liftE : (A → B) → SF (E A) (E B)
liftE f ≈ mapE f
```

$$
\begin{aligned}
&\mathit{liftC2} \;:\; (A \to B \to Z) \to \mathit{SF}\ (\mathsf{C}\ A, \mathsf{C}\ B)\ (\mathsf{C}\ Z)\\
&\mathit{liftC2}\ f \;\approx\; \mathit{uncurry}\ (\mathit{mapC2}\ f)\\
&\mathit{liftS2} \;:\; (A \to B \to Z) \to \mathit{SF}\ (\mathsf{S}\ A, \mathsf{S}\ B)\ (\mathsf{S}\ Z)\\
&\mathit{liftS2}\ f \;\approx\; \mathit{uncurry}\ (\mathit{mapS2}\ f)\\
&\mathit{merge} \;:\; (A \to Z) \to (B \to Z) \to (A \to B \to Z) \to \mathit{SF}\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\\
&\mathit{merge}\ f_a\ f_b\ f_{ab} \approx \mathit{uncurry}\ (\mathit{mergeE2}\ f_a\ f_b\ f_{ab})\\
&\mathit{join} \;:\; (A \to B \to Z) \to \mathit{SF}\ (\mathsf{E}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\\
&\mathit{join}\ f \approx \mathit{uncurry}\ (\mathit{joinE2}\ f)\\
&\mathit{sampleWithC} \;:\; (A \to B \to Z) \to \mathit{SF}\ (\mathsf{C}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\\
&\mathit{sampleWithC}\ f \;\approx\; \mathit{uncurry}\ (\mathit{mapCE}\ f)\\
&\mathit{sampleWithS} \;:\; (A \to B \to Z) \to \mathit{SF}\ (\mathsf{S}\ A, \mathsf{E}\ B)\ (\mathsf{E}\ Z)\\
&\mathit{sampleWithS}\ f \;\approx\; \mathit{uncurry}\ (\mathit{mapSE}\ f)
\end{aligned}
$$

B.3 Delaying Signals

The definition of *delayC* was given in Section 4.2.4. Thus this section only considers *delayE* and *delayS*.

Delaying a change list is just a matter of increasing the first time delta:

$$
\begin{aligned}
&\mathit{delayCL} \;:\; \mathit{Time} \to \mathit{ChangeList}\ A \to \mathit{ChangeList}\ A\\
&\mathit{delayCL}\ d\ [\,] &&= [\,]\\
&\mathit{delayCL}\ d\ ((\delta, a) :: \delta as) &&= (d + \delta, a) :: \delta as
\end{aligned}
$$

A useful variant of this function (optionally) takes a value as an additional argument and inserts it as a change at the delay time in the resultant change list:

$$
\begin{aligned}
&\mathit{delayCLinit} \;:\; \mathit{Maybe}\ A \to \mathit{Time}^{+} \to \mathit{ChangeList}\ A \to \mathit{ChangeList}\ A\\
&\mathit{delayCLinit}\ (\mathsf{just}\ a)\ d\ \delta as \;=\; (d, a) :: \delta as\\
&\mathit{delayCLinit}\ \mathsf{nothing}\ d\ \delta as \;=\; \mathit{delayCL}\ d\ \delta as
\end{aligned}
$$

Delaying a change prefix is achieved by reducing the sample time by the delay period (d), and then delaying the resultant change list by that amount:

$$
\begin{aligned}
&\mathit{delayCP} \;:\; \mathit{Maybe}\ A \to \mathit{Time}^{+} \to \mathit{ChangePrefix}\ A \to \mathit{ChangePrefix}\ A\\
&\mathit{delayCP}\ ma\ d\ cp\ t \mid t < d \;=\; [\,]\\
&\phantom{\mathit{delayCP}\ ma\ d\ cp\ t} \mid t \geqslant d \;=\; \mathit{delayCLinit}\ ma\ d\ (cp\ (t - d))
\end{aligned}
$$

Defining the *delay* signal functions is now straightforward:

$$
\begin{aligned}
&\mathit{delayE} \;:\; \mathit{Time}^{+} \to \mathit{SF}\ (\mathsf{E}\ A)\ (\mathsf{E}\ A)\\
&\mathit{delayE}\ d \approx \lambda\ (ma, cp) \to (\mathsf{nothing}, \mathit{delayCP}\ ma\ d\ cp)\\[1ex]
&\mathit{delayS} \;:\; \mathit{Time}^{+} \to A \to \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{S}\ A)\\
&\mathit{delayS}\ d\ a_0 \approx \lambda\ (a_1, cp) \to (a_0, \mathit{delayCP}\ (\mathsf{just}\ a_1)\ d\ cp)
\end{aligned}
$$

B.4 Filtering Event Signals

As an aid to filtering event signals, a function to filter change lists is defined as follows:

$$
\begin{aligned}
&\mathit{filterCL} \;:\; (A \to \mathit{Bool}) \to \mathit{ChangeList}\ A \to \mathit{ChangeList}\ A\\
&\mathit{filterCL}\ p\ [\,] &&= [\,]\\
&\mathit{filterCL}\ p\ ((\delta, a) :: \delta as) &&= \mathbf{if}\ p\ a\\
& &&\quad\ \mathbf{then}\ (\delta, a) :: \mathit{filterCL}\ p\ \delta as\\
& &&\quad\ \mathbf{else}\ \mathit{delayCL}\ \delta\ (\mathit{filterCL}\ p\ \delta as)
\end{aligned}
$$

Note that the time delta that follows any eliminated event occurrences must be increased (using *delayCL*). The definition of *filterE* is then as follows:

$$
\begin{aligned}
&\mathit{filterE} \;:\; (A \to \mathit{Bool}) \to \mathit{SF}\;(\mathsf{E}\;A)\;(\mathsf{E}\;A)\\
&\mathit{filterE}\;p \approx \lambda\,(\mathit{ma}, \mathit{cp}) \to (\mathit{maybeFilter}\;p\;\mathit{ma}, \mathit{result}\;(\mathit{filterCL}\;p)\;\mathit{cp})
\end{aligned}
$$

B.5 Dynamic Combinators

The conceptual definitions of the *switch* and *freeze* combinators are fairly involved. As an aid to defining them, several auxiliary notions will first be introduced.

B.5.1 Advancing Signals

Advancing a signal is the opposite of delaying a signal (some authors call it *ageing* the signal). That is, where delaying a signal looks into the past, advancing a signal looks into the future. Intuitively, advancing a signal shifts the local time frame of a signal forwards by a given amount of time (d), discarding everything before time d. This is acausal, and so wouldn't make sense as a signal function. However, *advance* is only used as a conceptual utility by *switch*, connecting a signal from outside the switching combinator to the local time of the residual signal function (because the time frame of the external network will be ahead of the local time frame of the residual signal function).

First, a change list can be advanced as follows:

$$
\begin{aligned}
&\mathit{advanceCL} \;:\; \mathit{Time} \to \mathit{ChangeList}\;A \to \mathit{ChangeList}\;A\\
&\mathit{advanceCL}\;d\;[\,] \;=\; [\,]\\
&\mathit{advanceCL}\;d\;((\delta, a) :: \delta as) \;\mid\; d < \delta \;=\; (\delta - d, a) :: \delta as\\
&\qquad\qquad\qquad\qquad\qquad\quad\;\mid\; d \geqslant \delta \;=\; \mathit{advanceCL}\;(d - \delta)\;\delta as
\end{aligned}
$$

Advancing a change prefix is achieved by sampling the prefix in the future, then advancing the resultant change list:

$$
\begin{aligned}
&\mathit{advanceCP} \;:\; \mathit{Time} \to \mathit{ChangePrefix}\;A \to \mathit{ChangePrefix}\;A\\
&\mathit{advanceCP}\;d\;\mathit{cp}\;t \;=\; \mathit{advanceCL}\;d\;(\mathit{cp}\;(t + d))
\end{aligned}
$$

Advancing a signal vector is then defined as follows:

$$
\begin{aligned}
&\mathit{advance} \;:\; \{\mathit{as} \;:\; \mathit{SVDesc}\} \to \mathit{Time} \to \mathit{SigVec}\;\mathit{as} \to \mathit{SigVec}\;\mathit{as}\\
&\mathit{advance}\;\{\mathsf{C}\;\_\}\;d\;s &&= \lambda\,t \to s\;(t + d)\\
&\mathit{advance}\;\{\mathsf{S}\;\_\}\;d\;s &&= (\mathit{valS}\;s\;d, \mathit{advanceCP}\;d\;(\mathit{snd}\;s))\\
&\mathit{advance}\;\{\mathsf{E}\;\_\}\;d\;s &&= (\mathit{occ}\;s\;d, \mathit{advanceCP}\;d\;(\mathit{snd}\;s))\\
&\mathit{advance}\;\{\_,\_\}\;d\;(s_1, s_2) &&= (\mathit{advance}\;d\;s_1, \mathit{advance}\;d\;s_2)
\end{aligned}
$$

B.5.2 Splicing

Switching combinators switch-out signal functions at certain time points, and replace them with newly switched-in signal functions. To describe this conceptually, a notion of *temporally composing* signals is needed. This is referred to as *splicing* signals.

First, as an aid to computing the earlier half of a splice, an auxiliary function is needed that takes a change prefix up to (yet excluding) a time point. For convenience, this function also returns the time delta between the time point and the last change in the resultant change list:

```
takeExclEnd : ChangePrefix A → $Time^+$ → ChangeList A × $\Delta t$
takeExclEnd cp t = let $\delta as$ = takeExcl t (cp t) in ($\delta as$, t − lastChangeTime $\delta as$)
```

Splicing signal vectors is then defined as follows:

```
spliceC : SigVec (C A) → SigVec (C A) → Time → SigVec (C A)
spliceC $s_1$ $s_2$ $t_x$ t | t < $t_x$ = $s_1$ t
                  | t ⩾ $t_x$ = $s_2$ (t − $t_x$)

spliceS : SigVec (S A) → SigVec (S A) → Time → SigVec (S A)
spliceS ($a_1$, $cp_1$) ($a_2$, $cp_2$) $t_x$
   | $t_x$ ≡ 0 = ($a_2$, $cp_2$)
   | $t_x$ ⩾ 0 = ($a_1$, λ t → if t < $t_x$
                             then $cp_1$ t
                             else let ($\delta as$, δ) = takeExclEnd $cp_1$ $t_x$
                                  in $\delta as$ ++ (δ, $a_2$) :: $cp_2$ (t − $t_x$))

spliceE : SigVec (E A) → SigVec (E A) → Time → SigVec (E A)
spliceE ($ma_1$, $cp_1$) ($ma_2$, $cp_2$) $t_x$
   | $t_x$ ≡ 0 = ($ma_2$, $cp_2$)
   | $t_x$ ⩾ 0 = ($ma_1$, λ t → if t < $t_x$
                               then $cp_1$ t
                               else let ($\delta as$, δ) = takeExclEnd $cp_1$ $t_x$
                                    in $\delta as$ ++ delayCLinit $ma_2$ δ ($cp_2$ (t − $t_x$)))

splice : {as : SVDesc} → SigVec as → SigVec as → Time → SigVec as
splice {C _} $s_1$          $s_2$          t = spliceC $s_1$ $s_2$ t
splice {S _} $s_1$          $s_2$          t = spliceS $s_1$ $s_2$ t
splice {E _} $s_1$          $s_2$          t = spliceE $s_1$ $s_2$ t
splice {_, _} ($sa_1$, $sb_1$) ($sa_2$, $sb_2$) t = (splice $sa_1$ $sa_2$ t, splice $sb_1$ $sb_2$ t)
```

The time argument is the time point at which the splice should occur. Essentially, *splice* s_1 s_2 t takes the prefix of s_1 over the interval $[0, t)$ and appends it temporally in front of s_2.

B.5.3 Assuming the Sample Time

The final utility is a function that allows a signal vector to depend on the time at which it is sampled, even though that sample time is not yet known:

```
withTime : {as : SVDesc} → (SampleTime → SigVec as) → SigVec as
withTime {C _} f = λ t → f t t
withTime {E _} f = (fst (f 0), λ t → snd (f t) t)
withTime {S _} f = (fst (f 0), λ t → snd (f t) t)
withTime {_, _} f = (withTime (fst ∘ f), withTime (snd ∘ f))
```

This will be useful when defining *switch* because the resultant signal vector of *switch* depends on whether an event has occurred yet, but that cannot be determined unless the sample time is known.

B.5.4 Switch

The *switch* combinator can now be defined as follows:

```
switch : SF as (bs, E A) → (A → SF as bs) → SF as bs
switch sf f ≈ λ $s_a$ → let ($s_b$, $s_e$) = sf $s_a$
                    in withTime (λ t → case fstOcc $s_e$ t of
                                         nothing      → $s_b$
                                         just ($t_e$, e) → splice $s_b$ ((f e) (advance $t_e$ $s_a$)) $t_e$)
```

If no event has occurred then the output is the subordinate signal function's output (s_b). If an event has occurred, then its value is used to generate a residual signal function ($f\ e$). The input signal vector is then advanced by the event time (t_e), and the residual signal function applied to it. Finally, s_b and the newly generated signal vector are spliced together.

B.5.5 Freeze

The definition of *freeze* is somewhat simpler:

$$\begin{array}{l} \mathit{freeze} \ :\ \mathit{SF}\ \mathit{as}\ \mathit{bs} \rightarrow \mathit{SF}\ \mathit{as}\ (\mathit{bs}, \mathsf{C}\ (\mathit{SF}\ \mathit{as}\ \mathit{bs})) \\ \mathit{freeze}\ \mathit{sf} \approx \lambda\, s_1 \rightarrow (\mathit{sf}\ s_1, \lambda\, t \rightarrow (\lambda\, s_2 \rightarrow \mathit{advance}\ t\ (\mathit{sf}\ (\mathit{splice}\ s_1\ s_2\ t)))) \end{array}$$

B.6 Miscellaneous Signal Functions

The *fromS* and *dfromS* signal functions are straightforwardly defined using *val* and *leftLimit*:

$$\begin{array}{l} \mathit{fromS} \ :\ \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{C}\ A) \\ \mathit{fromS} \approx \mathit{val} \end{array}$$

$$\begin{array}{l} \mathit{dfromS} \ :\ A \rightarrow \mathit{SF}\ (\mathsf{S}\ A)\ (\mathsf{C}\ A) \\ \mathit{dfromS}\ a \approx \lambda\, s\ t \rightarrow \mathbf{if}\ t > 0\ \mathbf{then}\ \mathit{leftLimit}\ s\ t\ \mathbf{else}\ a \end{array}$$

The integral of a Step signal is defined exactly using the rectangle rule:

$$\begin{array}{l} \mathit{integralS} \ :\ \mathit{SF}\ (\mathsf{S}\ \mathbb{R})\ (\mathsf{C}\ \mathbb{R}) \\ \mathit{integralS} \approx \lambda\, (a_0, \mathit{cp})\ t \rightarrow \begin{array}[t]{l} \mathbf{let}\ \begin{array}[t]{lcl} \delta \mathit{as} & = & \mathit{cp}\ t \\ \delta s & = & \mathit{map}\ \mathit{fst}\ \delta \mathit{as} \mathbin{+\!\!+} (t - \mathit{lastChangeTime}\ \delta \mathit{as}) :: [\,] \\ \mathit{as} & = & a_0 :: \mathit{map}\ \mathit{snd}\ \delta \mathit{as} \end{array} \\ \mathbf{in}\ \mathit{sum}\ (\mathit{zipWith}\ (*)\ \delta s\ \mathit{as}) \end{array} \end{array}$$

B.7 Rising Edge Detection (*when*)

The following conceptual definition of *when* is inspired by Wan and Hudak's definition [127]. The key difference, beside an adaptation to the N-ary FRP model, is a direct characterisation of the conditions required for a temporal predicate to be sufficiently well-behaved to make the definition of *when* meaningful. In Wan and Hudak's definition, this is indirect from the lack of a solution satisfying their stated semantic conditions. Intuitively, a temporal predicate is well-behaved if the list of positive transitions (transitions from the predicate not holding to the predicate holding) over any given interval is finite.

First, a number of auxiliary temporal predicates are required. *Over* requires a temporal predicate to hold over an interval:

$$\begin{array}{l} \mathit{Over} \ :\ \mathit{TPred} \rightarrow \mathit{Interval} \rightarrow \mathit{Set} \\ \mathit{Over}\ \varphi\ i\ =\ \forall\ t \rightarrow t \in i \rightarrow \varphi\ t \end{array}$$

PIvl and *FIvl* require there to exist non-empty open intervals to the left or right (respectively) of the time point, over which φ holds:

$$\begin{array}{l} \mathit{PIvl} \ :\ \mathit{TPred} \rightarrow \mathit{TPred} \\ \mathit{PIvl}\ \varphi\ t\ =\ \mathbf{P}\ (\lambda\, t_0 \rightarrow \mathit{Over}\ \varphi\ \langle\, t_0, t\,\rangle)\ t \\ \mathit{FIvl} \ :\ \mathit{TPred} \rightarrow \mathit{TPred} \\ \mathit{FIvl}\ \varphi\ t\ =\ \mathbf{F}\ (\lambda\, t_1 \rightarrow \mathit{Over}\ \varphi\ \langle\, t, t_1\,\rangle)\ t \end{array}$$

In a similar vein, *Neighbourhood* φ holds if there exists a neighbourhood around the time point over which φ holds:

$$\begin{aligned}&\mathit{Neighbourhood} : \mathit{TPred} \rightarrow \mathit{TPred}\\ &\mathit{Neighbourhood}\ \varphi = \mathit{PIvl}\ \varphi \wedge \varphi \wedge \mathit{FIvl}\ \varphi\end{aligned}$$

Transitions can now be characterised as temporal predicates:

$$\begin{aligned}&\mathit{PosTrans} : \mathit{TPred} \rightarrow \mathit{TPred}\\ &\mathit{PosTrans}\ \varphi = \mathit{PIvl}\ (\neg\ \varphi) \wedge \varphi \wedge \mathit{FIvl}\ \varphi\\ &\mathit{NegTrans} : \mathit{TPred} \rightarrow \mathit{TPred}\\ &\mathit{NegTrans}\ \varphi = \mathit{PIvl}\ \varphi \wedge \mathit{FIvl}\ (\neg\ \varphi)\\ &\mathit{NoTrans} : \mathit{TPred} \rightarrow \mathit{TPred}\\ &\mathit{NoTrans}\ \varphi = \mathit{Neighbourhood}\ \varphi \vee \mathit{Neighbourhood}\ (\neg\ \varphi)\\ &\mathit{PosTransL} : \mathit{TPred} \rightarrow \mathit{TPred}\\ &\mathit{PosTransL}\ \varphi = \mathit{PIvl}\ (\neg\ \varphi) \wedge \varphi\end{aligned}$$

PosTrans φ t holds if φ has a positive transition at the time t, *NegTrans* φ t holds if φ has a negative transition at point t, and *NoTrans* φ t holds if φ has no transition at point t. Note that *NegTrans* is not concerned with whether φ holds *at* point t, whereas *PosTrans* is. This is an due to the N-ary FRP model only permitting events to occur *at* time points, not immediately afterwards (see Section 9.1). Because of this limitation, it is necessary to rule out positive transitions where the predicate holds immediately after, but not at, a time point. Finally, *PosTransL* is the left-biased version of *PosTrans* that only considers an interval to the left of t.

A predicate that holds if φ is well-behaved on an open interval (t_0, t_1) can now be defined:

$$\begin{aligned}&\mathit{WellBehaved} : \mathit{TPred} \rightarrow \mathit{Time} \rightarrow \mathit{Time} \rightarrow \mathit{Set}\\ &\mathit{WellBehaved}\ \varphi\ t_0\ t_1 = \mathbf{finite}\ \{\tau \mid \tau \in \langle\ t_0, t_1\ \rangle, \mathit{PosTrans}\ \varphi\ \tau\}\\ &\qquad\qquad \times\ \mathit{Over}\ (\mathit{PosTrans}\ \varphi \vee \mathit{NegTrans}\ \varphi \vee \mathit{NoTrans}\ \varphi)\ \langle\ t_0, t_1\ \rangle\end{aligned}$$

The finiteness condition rules out the temporal predicate oscillating infinitely often over a finite interval. The second part says that it must be possible to characterise every interior point either as a positive transition, a negative transition, or the absence of a transition. This rules out "spikes": points where the truth of the predicate differs from its truth in all neighbourhoods of that point.

The finite ascending list of time points of positive transitions for a temporal predicate φ over an interval $(0, t]$ can now be defined:

$$\begin{aligned}&\mathit{poccs} : \mathit{TPred} \rightarrow \mathit{Time} \rightarrow \mathit{List}\ \mathit{Time}\\ &\mathit{poccs}\ \varphi\ t \mid \mathit{WellBehaved}\ \varphi\ 0\ t = [\tau \mid \tau \in \langle\ 0, t\ \rangle, \mathit{PosTrans}\ \varphi\ \tau] +\!\!+ [t \mid \mathit{PosTransL}\ \varphi\ t]\end{aligned}$$

Note that the use of the proposition *WellBehaved* φ 0 t in the pattern guard is informal notation expressing that *poccs* is partial. Thus *poccs* is only defined for well-behaved temporal predicates.

Finally, *when* is defined using *poccs*. Thus *when* is undefined if applied to an ill-behaved predicate and signal composition:

```
when : (A → Bool) → SF (C A) (E A)
when p ≈ λ s → (nothing, whenAux s)
  where
    whenAux : (Time → A) → ChangePrefix A
    whenAux s t = let ts = poccs (isTrue ∘ p ∘ s) t
                  in zipWith (λ t₁ t₀ → (t₁ − t₀, s t₁)) ts (0 :: ts)
```

Appendix C

Source Code for Embeddings of N-ary FRP

Chapters 5 and 7 defined embedded implementations of N-ary FRP in Agda and Haskell. When describing those implementations, a substantial amount of code was omitted as it was either uninteresting, verbose, or similar to other code. This appendix contains that omitted code.

Note that standard library code is not included, though many of the utility functions used can be found in Appendix A. The complete source code is available in the online archive [1].

C.1 The Delay Primitives

The *delay* family of primitive signal functions was omitted from the Agda embedding in Section 5.2 because their definitions are substantially more extensive than the other primitives. This section contains those definitions. The encodings of the *delay* primitives in the other embeddings in this thesis are not given, but they are available in the online archive [1].

As discussed in Section 5.4, the basic idea is to put the input samples in a queue, and then dequeue each sample after the delay period has passed. A queue module that provides the following API is assumed:

```
Queue              : Set → Set
emptyQueue         : ∀ {A} → Queue A
enQueue            : ∀ {A} → A → Queue A → Queue A
deQueue            : ∀ {A} → Queue A → Maybe (Queue A × A)
deQueueIf          : ∀ {A} → (A → Bool) → Queue A → Maybe (Queue A × A)
deQueueWhileLast   : ∀ {A} → (A → Bool) → Queue A → Maybe (Queue A × A)
```

The *deQueueIf* function only dequeues the head of the queue if the predicate holds for the head element. The *deQueueWhileLast* function dequeues elements while the predicate holds, and returns the last such element (if any). The other functions are standard.

Using these functions, the *delay* signal functions are defined as follows:

```
private CurrentTime  =  Time
        ReleaseTime  =  Time

        DelayQueue : Set → Set
        DelayQueue A  =  Queue (ReleaseTime × A)
        ready : {A : Set} → CurrentTime → ReleaseTime × A → Bool
        ready ct (rt, _) = ct ⩾ rt

  delayC : ∀ {A} → Time+ → (Time → A) → SF (C A) (C A)
  delayC {A} d f = mkSF delayAuxC (λ a0 → ((0, nothing, enQueueC 0 a0 emptyQueue), f 0))
    where
        -- The "Maybe A" tells us whether we are still in the delay period (nothing)
        -- or not (just a, where a is the most recent output sample)
      DelayStateC = CurrentTime × Maybe A × DelayQueue A

      enQueueC : CurrentTime → A → DelayQueue A → DelayQueue A
      enQueueC t a = enQueue (t + d, a)

      deQueueC : CurrentTime → DelayQueue A → Maybe (DelayQueue A × ReleaseTime × A)
      deQueueC t = deQueueWhileLast (ready t)

      deQueueCstate : CurrentTime → Maybe A → DelayQueue A → Maybe A × DelayQueue A × A
      deQueueCstate t st q with deQueueC t q
      deQueueCstate t st q         | just (q′, (_, a2)) = (just a2, q′, a2)
      deQueueCstate t nothing q    | nothing = (nothing, q, f t)
      deQueueCstate t (just a1) q  | nothing = (just a1, q, a1)

      delayAuxTimeC : CurrentTime → Maybe A → DelayQueue A → A → DelayStateC × A
      delayAuxTimeC t ma1 q a with deQueueCstate t ma1 q
      ... | (ma2, q′, a2) = ((t, ma2, enQueueC t a q′), a2)

      delayAuxC : Δt → DelayStateC → A → DelayStateC × A
      delayAuxC δ (t1, ma1, q) = delayAuxTimeC (δ + t1) ma1 q

  delayS : ∀ {A} → Time+ → A → SF (S A) (S A)
  delayS {A} d a0 = mkSF delayAuxS (λ a1 → ((0, a0, enQueueS 0 a1 emptyQueue), a0))
    where
      DelayStateS = CurrentTime × A × DelayQueue A

      enQueueS : CurrentTime → A → DelayQueue A → DelayQueue A
      enQueueS t a = enQueue (t + d, a)

      deQueueS : CurrentTime → A → DelayQueue A → DelayQueue A × A
      deQueueS t a1 q with deQueueWhileLast (ready t) q
      ... | nothing          = (q, a1)
      ... | just (q′, (_, a2)) = (q′, a2)

      delayAuxTimeS : CurrentTime → A → DelayQueue A → A → DelayStateS × A
      delayAuxTimeS t a1 q a with deQueueS t a1 q
      ... | (q′, a2) = ((t, a2, enQueueS t a q′), a2)

      delayAuxS : Δt → DelayStateS → A → DelayStateS × A
      delayAuxS δ (t1, a1, q) = delayAuxTimeS (δ + t1) a1 q

  delayE : ∀ {A} → Time+ → SF (E A) (E A)
  delayE {A} d = mkSF delayAuxE (λ e → ((0, enQueueE 0 e emptyQueue), noEvent))
    where
      DelayStateE = CurrentTime × DelayQueue A

      Event = Maybe

      enQueueE : CurrentTime → Event A → DelayQueue A → DelayQueue A
      enQueueE t nothing = id
      enQueueE t (just a) = enQueue (t + d, a)

      deQueueE : CurrentTime → DelayQueue A → DelayQueue A × Event A
      deQueueE t q with deQueueIf (ready t) q
      ... | nothing          = (q, noEvent)
      ... | just (q′, (_, a)) = (q′, event a)

      delayAuxTimeE : CurrentTime → DelayQueue A → Event A → DelayStateE × Event A
      delayAuxTimeE t q e with deQueueE t q
      ... | (q′, e) = ((t, enQueueE t e q′), e)
```

```
delayAuxE : Δt → DelayStateE → Event A → DelayStateE × Event A
delayAuxE δ (t₁, q) = delayAuxTimeE (δ + t₁) q
```

C.2 Haskell Embedding of N-ary FRP

In Section 5.3 much of the code for the Haskell embedding of N-ary FRP was omitted as it is very similar to the Agda code. The omitted code is listed in this section.

```
data Node :: ∗ → ∗ → ∗ where
  Node :: (Dt → q → Sample as → (q, Sample bs)) → q → Node as bs

stepNode :: Dt → Node as bs → Sample as → (Node as bs, Sample bs)
stepNode dt (Node f q) sa = first (Node f) (f dt q sa)

data AtomicRouter :: ∗ → ∗ → ∗ where
  SFId :: AtomicRouter as as
  Fst  :: AtomicRouter (as, bs) as
  Snd  :: AtomicRouter (as, bs) bs

stepRouter :: AtomicRouter as bs → Sample as → Sample bs
stepRouter SFId sa       = sa
stepRouter Fst  (sa1, _) = sa1
stepRouter Snd  (_, sa2) = sa2

data SF :: ∗ → ∗ → ∗ where
  Prim    :: (Sample as → (Node as bs, Sample bs)) → SF as bs
  ARouter :: AtomicRouter as bs                    → SF as bs
  Seq     :: SF as bs → SF bs cs                   → SF as cs
  Fan     :: SF as bs → SF as cs                   → SF as (bs, cs)
  Switch  :: SF as (bs, E e) → (e → SF as bs)      → SF as bs
  Freeze  :: SF as bs                              → SF as (bs, C (SF as bs))

data SF′ :: ∗ → ∗ → ∗ where
  Prim′    :: Node as bs                        → SF′ as bs
  ARouter′ :: AtomicRouter as bs                → SF′ as bs
  Seq′     :: SF′ as bs → SF′ bs cs             → SF′ as cs
  Fan′     :: SF′ as bs → SF′ as cs             → SF′ as (bs, cs)
  Switch′  :: SF′ as (bs, E e) → (e → SF as bs) → SF′ as bs
  Freeze′  :: SF′ as bs                         → SF′ as (bs, C (SF as bs))

step0 :: SF as bs → Sample as → (SF′ as bs, Sample bs)
step0 (Prim f) sa = first Prim′ (f sa)
step0 (ARouter r) sa = (ARouter′ r, stepRouter r sa)
step0 (Seq sf1 sf2) sa = let (sf1′, sb) = step0 sf1 sa
                             (sf2′, sc) = step0 sf2 sb
                         in (Seq′ sf1′ sf2′, sc)
step0 (Fan sf1 sf2) sa = let (sf1′, sb) = step0 sf1 sa
                             (sf2′, sc) = step0 sf2 sa
                         in (Fan′ sf1′ sf2′, (sb, sc))
step0 (Switch sf f) sa = case step0 sf sa of
                           (sf′, (sb, Nothing)) → (Switch′ sf′ f, sb)
                           (_, (_, Just e))     → step0 (f e) sa
step0 (Freeze sf) sa = let (sf′, sb) = step0 sf sa
                       in (Freeze′ sf′, (sb, sf))
```

```
step' :: Dt → SF' as bs → Sample as → (SF' as bs, Sample bs)
step' dt (Prim' n) sa = first Prim' (stepNode dt n sa)
step' dt (ARouter' r) sa = (ARouter' r, stepRouter r sa)
step' dt (Seq' sf1 sf2) sa = let (sf1', sb) = step' dt sf1 sa
                                 (sf2', sc) = step' dt sf2 sb
                             in (Seq' sf1' sf2', sc)
step' dt (Fan' sf1 sf2) sa = let (sf1', sb) = step' dt sf1 sa
                                 (sf2', sc) = step' dt sf2 sa
                             in (Fan' sf1' sf2', (sb, sc))
step' dt (Switch' sf f) sa = case step' dt sf sa of
                               (sf', (sb, Nothing)) → (Switch' sf' f, sb)
                               (_, (_, Just e))     → step0 (f e) sa
step' dt (Freeze' sf) sa = let (sf', sb) = step' dt sf sa
                           in (Freeze' sf', (sb, freezeSF dt sf))
  where
    freezeSF :: Dt → SF' as bs → SF as bs
    freezeSF dt (Prim' n)       = Prim (stepNode dt n)
    freezeSF dt (ARouter' r)    = ARouter r
    freezeSF dt (Seq' sf1 sf2)  = Seq (freezeSF dt sf1) (freezeSF dt sf2)
    freezeSF dt (Fan' sf1 sf2)  = Fan (freezeSF dt sf1) (freezeSF dt sf2)
    freezeSF dt (Switch' sf f)  = Switch (freezeSF dt sf) f
    freezeSF dt (Freeze' sf)    = Freeze (freezeSF dt sf)

mkSF :: (Dt → q → Sample as → (q, Sample bs)) → (Sample as → (q, Sample bs)) → SF as bs
mkSF f g = Prim (first (Node f).g)

mkSFsource :: (Dt → q → (q, Sample bs)) → q → Sample bs → SF as bs
mkSFsource f q sb = mkSF (λdt q' _ → f dt q') (const (q, sb))

mkSFtimeless :: (q → Sample as → (q, Sample bs)) → q → SF as bs
mkSFtimeless f q = mkSF (const f) (f q)

mkSFstateless :: (Sample as → Sample bs) → SF as bs
mkSFstateless f = mkSFtimeless (λ_ sa → ((), f sa)) ()

mkSFchangeless :: Sample bs → SF as bs
mkSFchangeless sb = mkSFstateless (const sb)

noEvent :: Sample (E a)
noEvent = Nothing

event :: a → Sample (E a)
event = Just

identity :: SF as as
identity = ARouter SFId

sfFst :: SF (as, bs) as
sfFst = ARouter Fst

sfSnd :: SF (as, bs) bs
sfSnd = ARouter Snd

(⋙) :: SF as bs → SF bs cs → SF as cs
(⋙) = Seq

(&&&) :: SF as bs → SF as cs → SF as (bs, cs)
(&&&) = Fan

switch :: SF as (bs, E e) → (e → SF as bs) → SF as bs
switch = Switch

freeze :: SF as bs → SF as (bs, C (SF as bs))
freeze = Freeze

constantS :: a → SF as (S a)
constantS = mkSFchangeless
```

```
never :: SF as (E a)
never = mkSFchangeless noEvent

now :: SF as (E ())
now = mkSFsource (λ_ _ → ((), noEvent)) () (event ())

notYet :: SF (E a) (E a)
notYet = mkSF (λ_ → curry id) (const ((), noEvent))

filterE :: (a → Bool) → SF (E a) (E a)
filterE p = mkSFstateless (maybeFilter p)

hold :: a → SF (E a) (S a)
hold = mkSFtimeless (λq → fork.fromMaybe q)

edge :: SF (S Bool) (E ())
edge = mkSFtimeless (λq i → (i, (if i && not q then event () else noEvent))) True

when :: (a → Bool) → SF (C a) (E a)
when p = mkSFtimeless (λq i → (p i, (if p i && not q then event i else noEvent))) True


type IntegralState = (Double, Double)

integrateRectangle :: Dt → IntegralState → Double → (IntegralState, Double)
integrateRectangle dt (tot, x1) x2 =  let tot′ = tot + (dt ∗ x1)
                                      in ((tot′, x2), tot′)

integrateTrapezium :: Dt → IntegralState → Double → (IntegralState, Double)
integrateTrapezium dt (tot, x1) x2 = let tot′ = tot + (dt ∗ (x1 + x2) / 2)
                                     in ((tot′, x2), tot′)

integralS :: SF (S Double) (C Double)
integralS = mkSF integrateRectangle (λx0 → ((0, x0), 0))

integralC :: SF (C Double) (C Double)
integralC = mkSF integrateTrapezium (λx0 → ((0, x0), 0))


liftC :: (a → b) → SF (C a) (C b)
liftC = mkSFstateless

liftS :: (a → b) → SF (S a) (S b)
liftS = mkSFstateless

liftE :: (a → b) → SF (E a) (E b)
liftE = mkSFstateless.fmap

liftC2 :: (a → b → z) → SF (C a, C b) (C z)
liftC2 = mkSFstateless.uncurry

liftS2 :: (a → b → z) → SF (S a, S b) (S z)
liftS2 = mkSFstateless.uncurry

merge :: (a → z) → (b → z) → (a → b → z) → SF (E a, E b) (E z)
merge fa fb fab = mkSFstateless (uncurry (maybeMerge fa fb fab))

join :: (a → b → z) → SF (E a, E b) (E z)
join = mkSFstateless.uncurry.liftM2

sampleWithC :: (a → b → z) → SF (C a, E b) (E z)
sampleWithC f = mkSFstateless (uncurry (fmap.f))

sampleWithS :: (a → b → z) → SF (S a, E b) (E z)
sampleWithS f = mkSFstateless (uncurry (fmap.f))


fromS :: SF (S a) (C a)
fromS = mkSFstateless id

dfromS :: a → SF (S a) (C a)
dfromS = mkSFtimeless (flip (,))
```

C.3 Agda Embedding of N-ary FRP with Feedback

This section contains the source code for the Agda embedding of N-ary FRP with Feedback (Section 7.5). Code that is unmodified from the embedding in Section 5.2 is not repeated.

```
data Node (as bs : SVDesc) : Dec → Set where
  cnode : ∀ { Q } → (Δt → Q → Sample as → Q × Sample bs)   → Q → Node as bs cau
  dnode : ∀ { Q } → (Δt → Q → (Sample as → Q) × Sample bs) → Q → Node as bs dec

stepNode : ∀ { as bs d } → Δt → Node as bs d → Sample as → Node as bs d × Sample bs
stepNode δ (cnode f q) sa = first (cnode f) (f δ q sa)
stepNode δ (dnode f q) sa = first (λ g → dnode f (g sa)) (f δ q)

dstepNode : ∀ { as bs } → Δt → Node as bs dec → (Sample as → Node as bs dec) × Sample bs
dstepNode δ (dnode f q) = first (λ g sa → dnode f (g sa)) (f δ q)

data SF : SVDesc → SVDesc → Dec → Set where
  cprim     : ∀ { as bs } → (Sample as → Node as bs cau × Sample bs)   → SF as bs cau
  dprim     : ∀ { as bs } → (Sample as → Node as bs dec) → Sample bs → SF as bs dec
  arouter   : ∀ { as bs } → AtomicRouter as bs                         → SF as bs cau
  seq       : ∀ { d1 d2 as bs cs } → SF as bs d1 → SF bs cs d2         → SF as cs (d1 ∨ d2)
  fan       : ∀ { d1 d2 as bs cs } → SF as bs d1 → SF as cs d2         → SF as (bs, cs) (d1 ∧ d2)
  rswitcher : ∀ { d1 d2 as bs A } → SF as (bs, E A) d1 → (A → SF as (bs, E A) d2) → SF as bs (d1 ∧ d2)
  freezer   : ∀ { d as bs } → SF as bs d                               → SF as (bs, C (SF as bs d)) d
  looper    : ∀ { d as bs cs } → SF (as, cs) bs d → SF bs cs dec       → SF as bs d
  weakener  : ∀ { d as bs } → SF as bs d                               → SF as bs cau

data SF′ : SVDesc → SVDesc → Dec → Set where
  prim      : ∀ { d as bs } → Node as bs d                        → SF′ as bs d
  arouter   : ∀ { as bs } → AtomicRouter as bs                    → SF′ as bs cau
  seq       : ∀ { d1 d2 as bs cs } → SF′ as bs d1 → SF′ bs cs d2  → SF′ as cs (d1 ∨ d2)
  fan       : ∀ { d1 d2 as bs cs } → SF′ as bs d1 → SF′ as cs d2  → SF′ as (bs, cs) (d1 ∧ d2)
  rswitcher : ∀ { d1 d2 as bs A } → SF′ as (bs, E A) d1 → (A → SF as (bs, E A) d2) → SF′ as bs (d1 ∧ d2)
  freezer   : ∀ { d as bs } → SF′ as bs d                         → SF′ as (bs, C (SF as bs d)) d
  looper    : ∀ { d as bs cs } → SF′ (as, cs) bs d → SF′ bs cs dec → SF′ as bs d
  weakener  : ∀ { d as bs } → SF′ as bs d                         → SF′ as bs cau

weakenSwitch : ∀ { as bs } → (d1 d2 : Dec) → SF′ as bs (d2 ∧ d2) → SF′ as bs (d1 ∧ d2)
weakenSwitch cau _     = weakener
weakenSwitch dec cau   = id
weakenSwitch dec dec   = id

mutual
  step0 : ∀ { d as bs } → SF as bs d → Sample as → SF′ as bs d × Sample bs
  step0 (cprim f) sa = first prim (f sa)
  step0 (dprim f sb) sa = (prim (f sa), sb)
  step0 (arouter r) sa = (arouter r, stepARouter r sa)
  step0 (seq sf1 sf2) sa with step0 sf1 sa
  ... | (sf1′, sb) with step0 sf2 sb
  ... | (sf2′, sc) = (seq sf1′ sf2′, sc)
  step0 (fan sf1 sf2) sa with step0 sf1 sa | step0 sf2 sa
  ... | (sf1′, sb) | (sf2′, sc) = (fan sf1′ sf2′, (sb, sc))
  step0 (rswitcher { d1 } { d2 } sf f) sa with step0 sf sa
  ... | (sf′, (sb, nothing)) = (rswitcher sf′ f, sb)
  ... | (_, (_, just e)) with step0 (f e) sa
  ... | (sf′, (sb, _)) = (weakenSwitch d1 d2 (rswitcher sf′ f), sb)
  step0 (freezer sf) sa with step0 sf sa
  ... | (sf′, sb) = (freezer sf′, (sb, sf))
  step0 (looper sff sfb) sa with dstep0 sfb
  ... | (g, sc) with step0 sff (sa, sc)
  ... | (sff', sb) = (looper sff' (g sb), sb)
  step0 (weakener sf) sa = first weakener (step0 sf sa)

  dstep0 : ∀ {as bs} → SF as bs dec → (Sample as → SF' as bs dec) × Sample bs
  dstep0 sf = dstepAux0 sf refl

  dstepAux0 : ∀ {d as bs} → SF as bs d → d ≡ dec → (Sample as → SF' as bs dec) × Sample bs
  dstepAux0 (cprim f) ()
  dstepAux0 (dprim f sb) refl = (prim ∘ f, sb)
  dstepAux0 (arouter r) ()
  dstepAux0 (seq {dec} sf1 sf2) refl with dstep0 sf1
  ... | (g, sb) with step0 sf2 sb
  ... | (sf2', sc) = ((λ sa → seq (g sa) sf2'), sc)
  dstepAux0 (seq {cau} {.dec} {as} {bs} {cs} sf1 sf2) refl with dstep0 sf2
  ... | (g, sc) = (aux, sc)
    where aux : Sample as → SF' as cs dec
          aux sa with step0 sf1 sa
          ... | (sf1', sb) = seq sf1' (g sb)
  dstepAux0 (fan {cau} sf1 sf2) ()
  dstepAux0 (fan {dec} sf1 sf2) refl with dstep0 sf1 | dstep0 sf2
  ... | (g1, sb) | (g2, sc) = ((λ sa → fan (g1 sa) (g2 sa)), (sb, sc))
  dstepAux0 (rswitcher {cau} sf f) ()
  dstepAux0 (rswitcher {dec} sf f) refl with dstep0 sf
  ... | (g, (sb, nothing)) = ((λ sa → rswitcher (g sa) f), sb)
  ... | (_, (_, just e)) with dstep0 (f e)
  ... | (g, (sb, _)) = ((λ sa → rswitcher (g sa) f), sb)
  dstepAux0 (freezer sf) refl with dstep0 sf
  ... | (g, sb) = (freezer ∘ g, (sb, sf))
  dstepAux0 (looper sff sfb) refl with dstep0 sff
  ... | (g, sb) with step0 sfb sb
  ... | (sfb', sc) = ((λ sa → looper (g (sa, sc)) sfb'), sb)
  dstepAux0 (weakener sf) ()

freezeSF : ∀ {d as bs} → Δt → SF' as bs d → SF as bs d
freezeSF δ (arouter r)        = arouter r
freezeSF δ (seq sf1 sf2)      = seq (freezeSF δ sf1) (freezeSF δ sf2)
freezeSF δ (fan sf1 sf2)      = fan (freezeSF δ sf1) (freezeSF δ sf2)
freezeSF δ (rswitcher sf f)   = rswitcher (freezeSF δ sf) f
freezeSF δ (freezer sf)       = freezer (freezeSF δ sf)
freezeSF δ (looper sff sfb)   = looper (freezeSF δ sff) (freezeSF δ sfb)
freezeSF δ (weakener sf)      = weakener (freezeSF δ sf)
freezeSF {cau} δ (prim n)     = cprim (stepNode δ n)
freezeSF {dec} δ (prim n)     = uncurry dprim (dstepNode δ n)

mutual
  step' : ∀ {d as bs} → Δt → SF' as bs d → Sample as → SF' as bs d × Sample bs
  step' δ (prim n) sa = first prim (stepNode δ n sa)
  step' δ (arouter r) sa = (arouter r, stepARouter r sa)
  step' δ (seq sf1 sf2) sa with step' δ sf1 sa
  ... | (sf1', sb) with step' δ sf2 sb
  ... | (sf2', sc) = (seq sf1' sf2', sc)
  step' δ (fan sf1 sf2) sa with step' δ sf1 sa | step' δ sf2 sa
  ... | (sf1', sb) | (sf2', sc) = (fan sf1' sf2', (sb, sc))
  step' δ (rswitcher {d1} {d2} sf f) sa with step' δ sf sa
  ... | (sf', (sb, nothing)) = (rswitcher sf' f, sb)
  ... | (_, (_, just e)) with step0 (f e) sa
  ... | (sf', (sb, _)) = (weakenSwitch d1 d2 (rswitcher sf' f), sb)
  step' δ (freezer sf) sa with step' δ sf sa
  ... | (sf', sb) = (freezer sf', (sb, freezeSF δ sf))
  step' δ (looper sff sfb) sa with dstep' δ sfb
  ... | (g, sc) with step' δ sff (sa, sc)
  ... | (sff', sb) = (looper sff' (g sb), sb)
  step' δ (weakener sf) sa = first weakener (step' δ sf sa)

  dstep' : ∀ {as bs} → Δt → SF' as bs dec → (Sample as → SF' as bs dec) × Sample bs
  dstep' δ sf = dstepAux' δ sf refl

  dstepAux' : ∀ {d as bs} → Δt → SF' as bs d → d ≡ dec
                → (Sample as → SF' as bs dec) × Sample bs
  dstepAux' δ (prim n) refl = (first ∘ result) prim (dstepNode δ n)
  dstepAux' δ (arouter r) ()
  dstepAux' δ (seq {dec} sf1 sf2) refl with dstep' δ sf1
  ... | (g, sb) with step' δ sf2 sb
  ... | (sf'2, sc) = ((λ sa → seq (g sa) sf'2), sc)
  dstepAux' δ (seq {cau} {_} {as} {_} {cs} sf1 sf2) refl with dstep' δ sf2
  ... | (g, sc) = (aux, sc)
              where aux : Sample as → SF' as cs dec
                aux sa with step' δ sf1 sa
                ... | (sf'1, sb) = seq sf'1 (g sb)
  dstepAux' δ (fan {cau} sf1 sf2) ()
  dstepAux' δ (fan {dec} sf1 sf2) refl with dstep' δ sf1 | dstep' δ sf2
  ... | (g1, sb) | (g2, sc) = ((λ sa → fan (g1 sa) (g2 sa)), (sb, sc))
  dstepAux' δ (rswitcher {cau} sf f) ()
  dstepAux' δ (rswitcher {dec} {cau} sf f) ()
  dstepAux' δ (rswitcher {dec} {dec} sf f) refl with dstep' δ sf
  ... | (g, (sb, nothing)) = ((λ sa → rswitcher (g sa) f), sb)
  ... | (_, (_, just e)) with dstep0 (f e)
  ... | (g, (sb, _)) = ((λ sa → rswitcher (g sa) f), sb)
  dstepAux' δ (freezer sf) refl with dstep' δ sf
  ... | (g, sb) = (freezer ∘ g, (sb, freezeSF δ sf))
  dstepAux' δ (looper sff sfb) refl with dstep' δ sff
  ... | (g, sb) with step' δ sfb sb
  ... | (sfb', sc) = ((λ sa → looper (g (sa, sc)) sfb'), sb)
  dstepAux' δ (weakener sf) ()

mkSFcau : ∀ {as bs Q} → (Δt → Q → Sample as → Q × Sample bs)
            → (Sample as → Q × Sample bs) → SF as bs cau
mkSFcau f g = cprim (first (cnode f) ∘ g)

mkSFdec : ∀ {as bs Q} → (Δt → Q → (Sample as → Q) × Sample bs)
            → (Sample as → Q) → Sample bs → SF as bs dec
mkSFdec f g = dprim (dnode f ∘ g)

mkSFsource : ∀ {as bs Q} → (Δt → Q → Q × Sample bs) → Q → Sample bs → SF as bs dec
mkSFsource f q = mkSFdec ((result2 ∘ first) const f) (const q)

mkSFtimeless : ∀ {as bs Q} → (Q → Sample as → Q × Sample bs) → Q → SF as bs cau
mkSFtimeless f q = mkSFcau (const f) (f q)

mkSFstateless : ∀ {as bs} → (Sample as → Sample bs) → SF as bs cau
mkSFstateless f = mkSFtimeless (curry (second f)) unit

mkSFchangeless : ∀ {as bs} → Sample bs → SF as bs dec
mkSFchangeless sb = mkSFsource (λ _ _ → (unit, sb)) unit sb
```

```
identity : ∀ {as} → SF as as cau
identity = arouter sfId

sfFst : ∀ {as bs} → SF (as, bs) as cau
sfFst = arouter fstProj

sfSnd : ∀ {as bs} → SF (as, bs) bs cau
sfSnd = arouter sndProj

_⋙_ : ∀ {d1 d2 as bs cs} → SF as bs d1 → SF bs cs d2 → SF as cs (d1 ∨ d2)
_⋙_ = seq

_&&&_ : ∀ {d1 d2 as bs cs} → SF as bs d1 → SF as cs d2 → SF as (bs, cs) (d1 ∧ d2)
_&&&_ = fan

rswitch : ∀ {d1 d2 as bs A} → SF as (bs, E A) d1 → (A → SF as (bs, E A) d2) → SF as bs (d1 ∧ d2)
rswitch = rswitcher

freeze : ∀ {d as bs} → SF as bs d → SF as (bs, C (SF as bs d)) d
freeze = freezer

loop : ∀ {d as bs cs} → SF (as, cs) bs d → SF bs cs dec → SF as bs d
loop = looper

weaken : ∀ {d as bs} → SF as bs d → SF as bs cau
weaken = weakener


constantS : ∀ {as A} → A → SF as (S A) dec
constantS = mkSFchangeless

never : ∀ {as A} → SF as (E A) dec
never = mkSFchangeless noEvent

now : ∀ {as} → SF as (E Unit) dec
now = mkSFsource (λ _ _ → (unit, noEvent)) unit (event unit)

notYet : ∀ {A} → SF (E A) (E A) cau
notYet = mkSFcau (λ _ → curry id) (const (unit, noEvent))

filterE : ∀ {A} → (A → Bool) → SF (E A) (E A) cau
filterE p = mkSFstateless (maybeFilter p)

hold : ∀ {A} → A → SF (E A) (S A) cau
hold = mkSFtimeless (λ q → fork ∘ fromMaybe q)

edge : SF (S Bool) (E Unit) cau
edge = mkSFtimeless (λ q i → (i, (if i && not q then event unit else noEvent))) true

when : ∀ {A} → (A → Bool) → SF (C A) (E A) cau
when p = mkSFtimeless (λ q i → (p i, (if p i && not q then event i else noEvent))) true


private
  IntegralState = ℝ × ℝ

  integrateRectangle : Δt → IntegralState → (ℝ → IntegralState) × ℝ
  integrateRectangle δ (tot, x1) = let tot′ = tot + (δ ∗ x1)
                                   in ((λ x2 → (tot′, x2)), tot′)

  integrateTrapezium : Δt → IntegralState → ℝ → IntegralState × ℝ
  integrateTrapezium δ (tot, x1) x2 = let tot′ = tot + (δ ∗ (x1 + x2) / 2)
                                      in ((tot′, x2), tot′)

integralS : SF (S ℝ) (C ℝ) dec
integralS = mkSFdec integrateRectangle (λ x0 → (0, x0)) 0

integralC : SF (C ℝ) (C ℝ) cau
integralC = mkSFcau integrateTrapezium (λ x0 → ((0, x0), 0))


liftC : ∀ {A B} → (A → B) → SF (C A) (C B) cau
liftC = mkSFstateless

liftS : ∀ {A B} → (A → B) → SF (S A) (S B) cau
liftS = mkSFstateless
```

```
liftE : ∀ {A B} → (A → B) → SF (E A) (E B) cau
liftE = mkSFstateless ∘ maybeMap

liftC2 : ∀ {A B Z} → (A → B → Z) → SF (C A, C B) (C Z) cau
liftC2 = mkSFstateless ∘ uncurry

liftS2 : ∀ {A B Z} → (A → B → Z) → SF (S A, S B) (S Z) cau
liftS2 = mkSFstateless ∘ uncurry

merge : ∀ {A B Z} → (A → Z) → (B → Z) → (A → B → Z) → SF (E A, E B) (E Z) cau
merge f_a f_b f_ab = mkSFstateless (uncurry (maybeMerge f_a f_b f_ab))

join : ∀ {A B Z} → (A → B → Z) → SF (E A, E B) (E Z) cau
join = mkSFstateless ∘ uncurry ∘ maybeMap2

sampleWithC : ∀ {A B Z} → (A → B → Z) → SF (C A, E B) (E Z) cau
sampleWithC f = mkSFstateless (uncurry (maybeMap ∘ f))

sampleWithS : ∀ {A B Z} → (A → B → Z) → SF (S A, E B) (E Z) cau
sampleWithS f = mkSFstateless (uncurry (maybeMap ∘ f))

fromS : ∀ {A} → SF (S A) (C A) cau
fromS = mkSFstateless id

dfromS : ∀ {A} → A → SF (S A) (C A) dec
dfromS = mkSFdec (λ _ q → (id, q)) id
```

C.4 Haskell Embedding of N-ary FRP with Feedback

Most of the source code for the Haskell embedding of N-ary FRP with Feedback (Section 7.6) was omitted. That omitted code can be found in this section. Code that is entirely unmodified from the embedding in Appendix C.2 is not repeated.

```
data Node :: ∗ → ∗ → ∗ → ∗ where
  CNode :: (Dt → q → Sample as → (q, Sample bs)) → q → Node as bs Cau
  DNode :: (Dt → q → ((Sample as → q), Sample bs)) → q → Node as bs Dec

stepNode :: Dt → Node as bs d → Sample as → (Node as bs d, Sample bs)
stepNode dt (CNode f q) sa = first (CNode f) (f dt q sa)
stepNode dt (DNode f q) sa = first (λg → DNode f (g sa)) (f dt q)

dstepNode :: Dt → Node as bs Dec → ((Sample as → Node as bs Dec), Sample bs)
dstepNode dt (DNode f q) = first (λg sa → DNode f (g sa)) (f dt q)

data SF :: ∗ → ∗ → ∗ → ∗ where
  CPrim   :: (Sample as → (Node as bs Cau, Sample bs))              → SF as bs Cau
  DPrim   :: (Sample as → Node as bs Dec) → Sample bs               → SF as bs Dec
  ARouter :: AtomicRouter as bs                                      → SF as bs Cau
  Seq     :: Decoupled d1 ⇒ SF as bs d1 → SF bs cs d2                → SF as cs (d1 ∨ d2)
  Fan     :: Decoupled d1 ⇒ SF as bs d1 → SF as cs d2                → SF as (bs, cs) (d1 ∧ d2)
  Switch  :: Decoupled d1 ⇒ SF as (bs, E e) d1 → (e → SF as bs d2)  → SF as bs (d1 ∧ d2)
  Freeze  :: SF as bs d                                               → SF as (bs, C (SF as bs d)) d
  Loop    :: SF (as, cs) bs d → SF bs cs Dec                          → SF as bs d
  Weaken  :: SF as bs d                                               → SF as bs Cau

data SF′ :: ∗ → ∗ → ∗ → ∗ where
  Prim′    :: Decoupled d ⇒ Node as bs d                                → SF′ as bs d
  ARouter′ :: AtomicRouter as bs                                        → SF′ as bs Cau
  Seq′     :: Decoupled d1 ⇒ SF′ as bs d1 → SF′ bs cs d2                → SF′ as cs (d1 ∨ d2)
  Fan′     :: Decoupled d1 ⇒ SF′ as bs d1 → SF′ as cs d2                → SF′ as (bs, cs) (d1 ∧ d2)
  Switch′  :: Decoupled d1 ⇒ SF′ as (bs, E e) d1 → (e → SF as bs d2)   → SF′ as bs (d1 ∧ d2)
  Freeze′  :: SF′ as bs d                                                → SF′ as (bs, C (SF as bs d)) d
  Loop′    :: SF′ (as, cs) bs d → SF′ bs cs Dec                          → SF′ as bs d
  Weaken′  :: SF′ as bs d                                                → SF′ as bs Cau
```

```
class Decoupled d where
  drep  :: SF as bs d → DRep d
  drepf :: (e → SF as bs d) → DRep d
  drep' :: SF' as bs d → DRep d
instance Decoupled Cau where
  drep  _ = Cau
  drepf _ = Cau
  drep' _ = Cau
instance Decoupled Dec where
  drep  _ = Dec
  drepf _ = Dec
  drep' _ = Dec

weakenSwitch :: DRep d1 → SF' as bs d2 → SF' as bs (d1 ∧ d2)
weakenSwitch Cau sf = Weaken' sf
weakenSwitch Dec sf = sf

step0 :: SF as bs d → Sample as → (SF' as bs d, Sample bs)
step0 (CPrim f) sa = first Prim' (f sa)
step0 (DPrim f sb) sa = (Prim' (f sa), sb)
step0 (ARouter r) sa = (ARouter' r, stepARouter r sa)
step0 (Seq sf1 sf2) sa = let (sf1', sb) = step0 sf1 sa
                             (sf2', sc) = step0 sf2 sb
                         in (Seq' sf1' sf2', sc)
step0 (Fan sf1 sf2) sa = let (sf1', sb) = step0 sf1 sa
                             (sf2', sc) = step0 sf2 sa
                         in (Fan' sf1' sf2', (sb, sc))
step0 (Switch sf f) sa = case step0 sf sa of
                           (sf', (sb, Nothing)) → (Switch' sf' f, sb)
                           (_, (_, Just e))     → first (weakenSwitch (drep sf)) (step0 (f e) sa)
step0 (Freeze sf) sa = let (sf', sb) = step0 sf sa
                       in (Freeze' sf', (sb, sf))
step0 (Loop sff sfb) sa = case dstep0 sfb of
                            (g, sc) → case step0 sff (sa, sc) of
                                        (sff', sb) → (Loop' sff' (g sb), sb)
step0 (Weaken sf) sa = first Weaken' (step0 sf sa)

dstep0 :: SF as bs Dec → ((Sample as → SF' as bs Dec), Sample bs)
dstep0 (DPrim f sb) = (Prim'.f, sb)
dstep0 (Seq sf1 sf2) = case drep sf1 of
                         Dec → let (g, sb)    = dstep0 sf1
                                   (sf2', sc) = step0 sf2 sb
                               in ((λsa → Seq' (g sa) sf2'), sc)
                         Cau → let (g, sc) = dstep0 sf2
                               in (λsa → let (sf1', sb) = step0 sf1 sa
                                         in Seq' sf1' (g sb)
                                  , sc)
dstep0 (Fan sf1 sf2) = case drep sf1 of
                         Dec → let (g1, sb) = dstep0 sf1
                                   (g2, sc) = dstep0 sf2
                               in ((λsa → Fan' (g1 sa) (g2 sa)), (sb, sc))
dstep0 (Switch sf f) = case drep sf of
                         Dec → case dstep0 sf of
                                 (g, (sb, Nothing)) → ((λsa → Switch' (g sa) f), sb)
                                 (_, (_, Just e))   → dstep0 (f e)
```

```
dstep0 (Freeze sf) = let (g, sb) = dstep0 sf
                     in (Freeze′.g, (sb, sf))
dstep0 (Loop sff sfb) = case dstep0 sff of
                            (g, sb) →  case step0 sfb sb of
                                         (sfb′, sc) → ((λsa → Loop′ (g (sa, sc)) sfb′), sb)

freezeSF :: Dt → SF′ as bs d → SF as bs d
freezeSF dt (ARouter′ r)    = ARouter r
freezeSF dt (Seq′ sf1 sf2)  = Seq (freezeSF dt sf1) (freezeSF dt sf2)
freezeSF dt (Fan′ sf1 sf2)  = Fan (freezeSF dt sf1) (freezeSF dt sf2)
freezeSF dt (Switch′ sf f)  = Switch (freezeSF dt sf) f
freezeSF dt (Freeze′ sf)    = Freeze (freezeSF dt sf)
freezeSF dt (Loop′ sff sfb) = Loop (freezeSF dt sff) (freezeSF dt sfb)
freezeSF dt (Weaken′ sf)    = Weaken (freezeSF dt sf)
freezeSF dt (Prim′ n)       = case n of
                                 CNode _ _ → CPrim (stepNode dt n)
                                 DNode _ _ → uncurry DPrim (dstepNode dt n)

step′ :: Dt → SF′ as bs d → Sample as → (SF′ as bs d, Sample bs)
step′ dt (Prim′ n) sa = first Prim′ (stepNode dt n sa)
step′ dt (ARouter′ r) sa = (ARouter′ r, stepARouter r sa)
step′ dt (Seq′ sf1 sf2) sa = let (sf1′, sb) = step′ dt sf1 sa
                                 (sf2′, sc) = step′ dt sf2 sb
                             in (Seq′ sf1′ sf2′, sc)
step′ dt (Fan′ sf1 sf2) sa = let (sf1′, sb) = step′ dt sf1 sa
                                 (sf2′, sc) = step′ dt sf2 sa
                             in (Fan′ sf1′ sf2′, (sb, sc))
step′ dt (Switch′ sf f) sa = case step′ dt sf sa of
                               (sf′, (sb, Nothing)) → (Switch′ sf′ f, sb)
                               (_, (_, Just e))      → first (weakenSwitch (drep′ sf)) (step0 (f e) sa)
step′ dt (Freeze′ sf) sa = let (sf′, sb) = step′ dt sf sa
                           in (Freeze′ sf′, (sb, freezeSF dt sf))
step′ dt (Loop′ sff sfb) sa = case dstep′ dt sfb of
                                (g, sc) → case step′ dt sff (sa, sc) of
                                             (sff′, sb) → (Loop′ sff′ (g sb), sb)
step′ dt (Weaken′ sf) sa = first Weaken′ (step′ dt sf sa)

dstep′ :: Dt → SF′ as bs Dec → ((Sample as → SF′ as bs Dec), Sample bs)
dstep′ dt (Prim′ n) = (first.result) Prim′ (dstepNode dt n)
dstep′ dt (Seq′ sf1 sf2) = case drep′ sf1 of
                             Dec → let (g, sb) = dstep′ dt sf1
                                           (sf2′, sc) = step′ dt sf2 sb
                                       in ((λsa → Seq′ (g sa) sf2′), sc)
                             Cau → let (g, sc) = dstep′ dt sf2
                                       in (λsa → let (sf1′, sb) = step′ dt sf1 sa
                                                  in  Seq′ sf1′ (g sb)
                                          , sc)
dstep′ dt (Fan′ sf1 sf2) = case drep′ sf1 of
                             Dec → let (g1, sb) = dstep′ dt sf1
                                           (g2, sc) = dstep′ dt sf2
                                       in ((λsa → Fan′ (g1 sa) (g2 sa)), (sb, sc))
dstep′ dt (Switch′ sf f) = case drep′ sf of
                             Dec → case dstep′ dt sf of
                                       (g, (sb, Nothing)) → ((λsa → Switch′ (g sa) f), sb)
                                       (_, (_, Just e))    → dstep0 (f e)
```

```
dstep′ dt (Freeze′ sf) = let (g, sb) = dstep′ dt sf
                         in (Freeze′.g, (sb, freezeSF dt sf))
dstep′ dt (Loop′ sff sfb) = case dstep′ dt sff of
                              (g, sb) → case step′ dt sfb sb of
                                          (sfb′, sc) → ((λsa → Loop′ (g (sa, sc)) sfb′), sb)

mkSFcau :: (Dt → q → Sample as → (q, Sample bs)) → (Sample as → (q, Sample bs)) → SF as bs Cau
mkSFcau f g = CPrim (first (CNode f).g)
mkSFdec :: (Dt → q → ((Sample as → q), Sample bs)) → (Sample as → q) → Sample bs → SF as bs Dec
mkSFdec f g = DPrim (DNode f.g)
mkSFsource :: (Dt → q → (q, Sample bs)) → q → Sample bs → SF as bs Dec
mkSFsource f q = mkSFdec ((result2.first) const f) (const q)
mkSFtimeless :: (q → Sample as → (q, Sample bs)) → q → SF as bs Cau
mkSFtimeless f q = mkSFcau (const f) (f q)
mkSFstateless :: (Sample as → Sample bs) → SF as bs Cau
mkSFstateless f = mkSFtimeless (curry (second f)) ()
mkSFchangeless :: Sample bs → SF as bs Dec
mkSFchangeless sb = mkSFsource (λ_ _ → ((), sb)) () sb

identity :: SF as as Cau
identity = ARouter SFId
sfFst :: SF (as, bs) as Cau
sfFst = ARouter Fst
sfSnd :: SF (as, bs) bs Cau
sfSnd = ARouter Snd
(⋙) :: Decoupled d1 ⇒ SF as bs d1 → SF bs cs d2 → SF as cs (d1 ∨ d2)
(⋙) = Seq
(&&&) :: Decoupled d1 ⇒ SF as bs d1 → SF as cs d2 → SF as (bs, cs) (d1 ∧ d2)
(&&&) = Fan
switch :: Decoupled d1 ⇒ SF as (bs, E e) d1 → (e → SF as bs d2) → SF as bs (d1 ∧ d2)
switch = Switch
freeze :: SF as bs d → SF as (bs, C (SF as bs d)) d
freeze = Freeze
loop :: SF (as, cs) bs d → SF bs cs Dec → SF as bs d
loop = Loop
weaken :: SF as bs d → SF as bs Cau
weaken = Weaken

constantS :: a → SF as (S a) Dec
constantS = mkSFchangeless
never :: SF as (E a) Dec
never = mkSFchangeless noEvent
now :: SF as (E ()) Dec
now = mkSFsource (λ_ _ → ((), noEvent)) () (event ())
notYet :: SF (E a) (E a) Cau
notYet = mkSFcau (λ_ → curry id) (const ((), noEvent))
filterE :: (a → Bool) → SF (E a) (E a) Cau
filterE p = mkSFstateless (maybeFilter p)
hold :: a → SF (E a) (S a) Cau
hold = mkSFtimeless (λq → fork.fromMaybe q)
edge :: SF (S Bool) (E ()) Cau
edge = mkSFtimeless (λq i → (i, (if i && not q then event () else noEvent))) True
when :: (a → Bool) → SF (C a) (E a) Cau
when p = mkSFtimeless (λq i → (p i, (if p i && not q then event i else noEvent))) True
```

```
type IntegralState = (Double, Double)

integrateRectangle :: Dt → IntegralState → ((Double → IntegralState), Double)
integrateRectangle dt (tot, x1) = let tot′ = tot + (dt ∗ x1)
                                  in ((λx2 → (tot′, x2)), tot′)

integrateTrapezium :: Dt → IntegralState → Double → (IntegralState, Double)
integrateTrapezium dt (tot, x1) x2 = let tot′ = tot + (dt ∗ (x1 + x2) / 2)
                                     in ((tot′, x2), tot′)

integralS :: SF (S Double) (C Double) Dec
integralS = mkSFdec integrateRectangle (λx0 → (0, x0)) 0

integralC :: SF (C Double) (C Double) Cau
integralC = mkSFcau integrateTrapezium (λx0 → ((0, x0), 0))


liftC :: (a → b) → SF (C a) (C b) Cau
liftC = mkSFstateless

liftS :: (a → b) → SF (S a) (S b) Cau
liftS = mkSFstateless

liftE :: (a → b) → SF (E a) (E b) Cau
liftE = mkSFstateless.fmap

liftC2 :: (a → b → z) → SF (C a, C b) (C z) Cau
liftC2 = mkSFstateless.uncurry

liftS2 :: (a → b → z) → SF (S a, S b) (S z) Cau
liftS2 = mkSFstateless.uncurry

merge :: (a → z) → (b → z) → (a → b → z) → SF (E a, E b) (E z) Cau
merge fa fb fab = mkSFstateless (uncurry (maybeMerge fa fb fab))

join :: (a → b → z) → SF (E a, E b) (E z) Cau
join = mkSFstateless.uncurry.liftM2

sampleWithC :: (a → b → z) → SF (C a, E b) (E z) Cau
sampleWithC f = mkSFstateless (uncurry (fmap.f))

sampleWithS :: (a → b → z) → SF (S a, E b) (E z) Cau
sampleWithS f = mkSFstateless (uncurry (fmap.f))


fromS :: SF (S a) (C a) Cau
fromS = mkSFstateless id

dfromS :: a → SF (S a) (C a) Dec
dfromS = mkSFdec (λ_ q → (id, q)) id
```