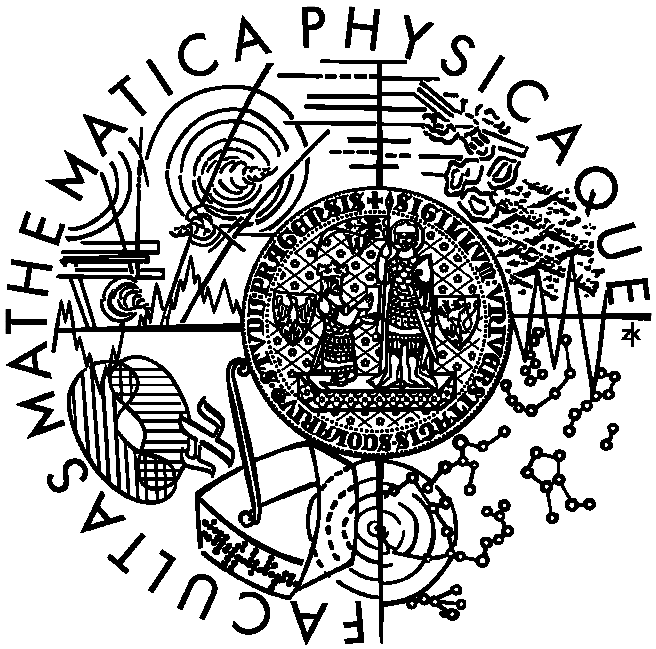
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Petříček

Reactive Programming with Events

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Don Syme

Study programme: Theoretical Computer Science

2010

I hereby certify that I wrote the thesis myself using only the referenced sources. I give consent with lending the thesis.

Prague, 15 April, 2010 Tomáš Petříček

Contents

Title: Reactive Programming with Events
Author: Tomáš Petříček
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Don Syme, Microsoft Research Cambridge
Supervisor's e-mail address: dsyme@microsoft.com

Abstract: The reactive programming model is largely different to what we're used to as we don't have a full control over the application's control flow. As a result, reactive applications need to be structured differently and need to be written using different patterns. We build an easier way for developing reactive applications. Our work integrates declarative and imperative approach to reactive programming and uses the notion of event as the unifying concept. We discuss the problem of memory management in this scenario and develop a technique for collecting not only events that are not reachable, but also events that cannot trigger any action. Next, we present a language extension that makes it possible to wait for occurrences of events that match some defined pattern. The extension isn't bound to the reactive programming model and can be used for concurrent and parallel programming scenarios as well. Finally, we develop a reactive programming library that builds on top of the ideas discussed earlier and we present semantics of the library to enable formal reasoning about reactive programs.

Keywords: reactive programming; monads; events; pattern matching

Název práce: Reactive Programming with Events
Autor: Tomáš Petříček
Katedra: Katedra teoretické informatiky a matematické logiky
Vedoucí práce: Don Syme, Microsoft Research Cambridge
E-mail vedoucího: dsyme@microsoft.com

Abstrakt: Programovací model pro reaktivní aplikace je v mnoha ohledech neobvyklý, protože nemáme plnou kontrolu nad řízením běhu aplikace. Z toho důvodu mají reaktivní aplikace jinou strukturu a musí být psány pomocí jiných návrhových vzorů. V této práci prezentujeme snazší způsob pro vývoj reaktivních aplikací. Naše práce propojuje deklarativní a imperativní přístup pomocí sjednocujícího konceptu událostí. V první části práce se zaměříme na správu paměti a prezentujeme techniku pro automatické uvolňování objektů reprezentujících událost nejen v situaci kdy událost není referencována z programu ale také v situaci kdy událost již nemůže způsobit žádnou reakci. Dále prezentujeme rozšíření jazyka, které umožňuje zápis kódu čekajícího na několik událostí. Toto rozšíření není pevně vázané na reaktivní programování a je tedy možné využít ho například i pro paralelní programování. Na závěr prezentujeme knihovnu pro reaktivní programování, která využívá všechny výše uvedené myšlenky a popíšeme sémantiku jazyka, která umožňuje formální dokazování vlastností reaktivních programů.

Keywords: reaktivní programování; monády; událost; porovnávání vzorů

Preface

I did most of the work presented in this thesis during an internship at Microsoft Research Cambridge between October 2008 and April 2009, which was supervised by Don Syme. I am very glad that Don was kind enough to supervise not only the internship, but also my work on this thesis. My interest in reactive programming dates back to 2007 when I worked on my Bachelor thesis (6) dealing with web programming. In fact, some examples that I will present later in this thesis look surprisingly alike to the Figure 11 from my Bachelor thesis.

The work presented in this thesis extends a relatively simple programming model that I developed with Don Syme during my internship. The model is presented in Chapter 16 of a book about functional programming that I wrote with the assistance of Jon Skeet (1). Since the publication, this programming model has attracted some attention by the F# community. I was also privileged enough to be invited to the Lang.NET conference at Microsoft campus to present this work (3), which provided me with a lot of valuable feedback and comments.

This thesis is partly based on articles that I wrote with Don Syme about our work. Some portions of Chapter III and most of the Chapter IV are based on the article (2), which was accepted for presentation at the International Symposium in Memory Management 2010 and will be published by ACM. The Chapter V is based on a yet unpublished draft (available on the attached CD). However an extended abstract based on the article advanced to the second round of Student Research Competition at the PLDI conference (4).

(1) T. Petricek with J. Skeet. Real-World Functional Programming With examples in F# and C#. ISBN: 978-1933988924. Manning, 2009.

(2) T. Petricek, D. Syme. Collecting Hollywood's Garbage: Avoiding Space-Leaks in Composite Events. To appear in Proceedings of ISMM 2010.

(3) T. Petricek, D. Syme. Reactive pattern matching for F#. Presented at Lang.NET 2009, Available at http://tinyurl.com/matchbang

(4) T. Petricek. Reactive and concurrent programming using pattern matching (extended abstract). Accepted at PLDI Student Research Competition 2010.

(5) T. Petricek. Client side scripting using meta-programming. Bachelor thesis, Charles University in Prague, 2007. Available at: http://tinyurl.com/fswebtools

Acknowledgements

First of all, I would like to thank to Don Syme for inviting me to an internship at Microsoft Research in Cambridge, for supervising my work during the internship and also for supervising this thesis. The shape of the work presented in this thesis has been formed during our long discussions by various whiteboards available at the Cambridge Lab and at The Green Man in Grantchester.

The Cambridge Lab has been a very inspiring place and I'm grateful to everyone who I had a chance to discuss the project with. Namely, I'd like to thank to James Margetson, Simon Peyton Jones, Claudio Russo and also many other people who provided interesting and useful comments during my presentation at the end of the internship.

I also received numerous useful comments during the presentation at the Lang.NET conference. I would like to thank Philip Wadler and others for asking the right questions. Discussions with the members of the F# language team and the Reactive Extensions for .NET team at Microsoft were another source for extremely valuable feedback. In particular, I would like to thank to Wes Dyer, Dmitry Lomov and Erik Meijer.

Chapter I

Introduction

A lot of the code that programmers write assumes that it's in the driving seat—that we're in control of what happens at each step. This is also the classical style of programming for which most of the languages and control-flow constructs were designed. Unfortunately, this model breaks down for many modern types of applications. For example, in *parallel programming*, computations are carried out simultaneously, *concurrent programming* models need to synchronize running processes and *reactive programming* requires waiting for events (such as user input), and acting in response.

The main focus of this thesis is on reactive programming models, but we also present some results that deal with parallel and concurrent programming and are, in general, more widely applicable. For the introduction, we'll start by focusing on reactive applications as they are intuitively easy to understand and demonstrate the problem nicely.

A typical Windows application needs to handle a variety of user interface events. Moreover, it may invoke asynchronous calls that instruct the operating system to perform some operation in background and notify the program of the result of the operation. The program may also perform some long-running computation on a background thread and once it completes, it needs to display the result in the user interface. In all these situations, the execution of this type of application is controlled by events, and the application is concerned with reacting to them. For this reason, this principle is sometimes called inversion of control and is sometimes lightheartedly referred to as *the Hollywood Principle*[1].

1. Standard event handling techniques

In most of the systems, event handling is implemented by some form of message loop that repeatedly receives a message from the system and handles the message (for example by modifying some program state stored in a global variable). In modern systems, the event loop is typically hidden from the user and messages are dispatched to a method of the application object, which is responsible for handling of one types of messages (such as `MouseClick`).

[1] "Don't call us, we'll call you"

Let's start by looking at an example. We want to implement an application displayed in Figure 1, which keeps some numeric state, displays the current state using a label and provides buttons for incrementing and decrementing the number.

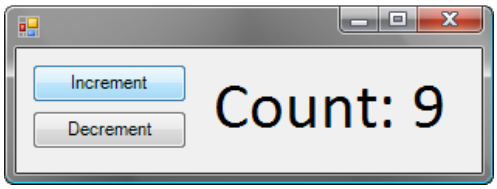


Figure 1. Counting the number of clicks

As most of the examples in this thesis, we'll write the code in the F# language [30], which targets the .NET platform. The language supports functional, imperative as well as object-oriented programming styles (discussed in more details in Chapter II). The following implementation uses the standard imperative and object-oriented approach:

```
type Program() =
  let mutable count = 0;

  // Omitted: initialization of the user interface

  member x.IncrementClick (sender:obj) (e:EventArgs) =
    count <- count + 1
    lblInfo.Text <- String.Format("Clicks: {0}", count)

  member x.DecrementClick (sender:obj) (e:EventArgs) =
    count <- count - count;
    lblInfo.Text <- String.Format("Clicks: {0}", count)
```

The code declares a class `Program` with a single mutable field `count` and two methods to handle `Click` events of the two buttons that are added to the main window. We could make the example nicer by declaring only one handler method and using the `sender` parameter to determine which of the buttons was clicked. However, code duplication isn't our primary concern.

This approach to dealing with events is limited in many ways. The following list gives several reasons that motivate our work:

- **Encapsulation of behavior patterns.** Suppose that we wanted to create another counter whose value would be incremented by a left button click and decremented by a right button click. Can we encapsulate the counting behavior into a reusable component?
- **Event interaction patterns.** Let's say we'd like to react to some combination of events. For example, we may want to react to the `Click` event only when the user previously pressed the Ctrl key (meaning that the `MouseDown` event was triggered and the `Key` property of the argument carried by the event had some specified value). Or, let's say that we'd like to react to the situation when the user presses a mouse button in one location, moves the mouse pointer to another location and releases the button. Is there some way to express these (and similar) interactions of events in a succinct and readable way?

- **Encoding state machines.** Many user interaction scenarios can be elegantly described as state machines. For example, when drawing a shape, there are two transitions. When the user presses Esc, she cancels the drawing which causes transition to the initial state. When she finishes drawing, the program transitions to the state in which it stores the newly drawn shape. Encoding this kind of program using a mutable state is painful, as we need a single mutable variable to hold the current state and numerous other variables to hold properties of individual states.
- **Referential transparency.** In functional programming, we avoid mutation, which has numerous benefits. For example, it makes programs easier to reason about, it simplifies unit-testing and it allows us to parallelize programs more easily. However, when using events in the style presented above, we rely on mutable state. Ideally, we'd like to write reactive programs that have the properties mentioned above.

In the following section, we'll give a high-level overview of the approach that we follow in this thesis and then we'll review our main contributions.

2. Approach and existing work

In this section, we'll give a brief overview of our approach to the problem. We don't aim to give a full and in depth overview here, we just present the background that is necessary for stating the main contributions of this work. The approach to the problem is in detail described in Chapter III and the existing work that we use as a basis for our work as well as the related work is in detail discussed in Chapter II.

2.1 Existing programming models

Techniques for writing reactive applications appear in different applications including synchronous languages from hardware programming, languages for programming robots and functional libraries for creating user-interface. In general, they can be categorized either as *declarative* (also called *data-flow*) or *imperative* (or *control-flow*).

The first approach is based on composing programs from primitives that represent some basic meaning or operation using combinators that compose primitives in some well defined way (for example, sequential or parallel composition). This approach is used in the synchronous language Lustre [10] and in techniques originating from Fran and functional reactive programming [1, 17].

When using the second approach, we encode programs as sequences of imperative actions. These actions may include emitting some signal or waiting for a signal (in hardware programming), repeating a sequence of operations until some condition holds or until something occurs or, for example, pausing the computation. This approach is used in the synchronous language Esterel [11, 12], in the Erlang [13] language (which is based on message-passing concurrency and the interesting operation is waiting for a message), but also in Haskell project named Imperative Streams [3] (which uses monads to make the computation sequential).

2.2 Reactive programming with events

In F#, we can program with events using the naïve imperative programming model described in section 1. In addition, the F# core library already provides a relatively limited support for the *declarative* (or *data-flow*) reactive programming model [19]. However, it didn't contain any advanced implementation of the *imperative* (or *control-flow*) programming model that would allow operations such as waiting for an event. We believe that both of the programming models have some benefits and that the best approach is to provide the user with both of the options and let her choose a model more appropriate for a specific task.

We provide an advanced library based on the *imperative* approach. The library uses F# computation expressions with a single extension that we design. The user can use operations such as waiting for a specified event and our extension makes it possible to wait for more complicated interaction patterns (such as a combination of `MouseDown` with a previous `KeyDown` event with the `Key` property set to a valuer representing the Ctrl key). This library also makes it possible to use many of the existing programming patterns that the users of functional language are familiar with such as encoding of state machines using mutually recursive functions.

Indeed, we don't want to provide two different programming models in a way, such that the user can use one or the other. As a result, we need to make it possible to combine parts of the application that are written using different techniques. We do this by building our library around the notion of an *event*, which is already used in the *declarative* model [19]. An event can be viewed as an asynchronous output channel of some running computation. The computation can trigger the event at some point and by sending some value to the channel. Other application's components can register with events to receive values from the channel. More formally, we can model events as a (possibly infinite) sequence of time/value pairs.

2.3 Expressing event interactions

As noted in section 1, we want to be able to express certain interactions of events in a succinct and readable way. Examples of the common patterns include waiting for the first of multiple events, waiting for all specified events and combinations such as waiting for first two of three provided events. This waiting should be embedded in the imperative programming model, so that it can be used as one of the operations that form the computation.

One way to solve these challenges is to design a specialized language, which provides a clear solution with the best possible syntax. However, this approach also binds the language to a single programming model. For our work this means that we would be designing language useful mainly for reactive programming, which is an important, but still a relatively limited area. This approach has been used by languages that are based on the Join calculus [31] such as JoCaml [32] and Cω [33]. These languages focus on the field of concurrent programming, but are in many ways closely related to our reactive programming model.

On the other hand, we can develop the solution solely as a library. This makes it possible to add the programming model to a widely used, general purpose programming language and it allows developers to freely choose the most appropriate programming model or even combine multiple models. However, the disadvantage

is that we are restricted by the syntax of the host language. Some examples of this approach from the concurrent programming field are encodings of Join calculus in C# [34], which relies on somewhat cumbersome syntax; the encoding of the same programming model in Scala [36], which uses extensible pattern matching in Scala to provide more elegant syntax. Another example is Concurrent ML [57], which provides concurrent programming model based on events[2]. As it is embedded in the ML language, it provides a functional DSL (domain specific language) for composing more complicated events.

An approach that lies between designing a new language and embedding the model as a library is to identify a repeating pattern in the library-based solutions and to provide a syntactic sugar that can be used by a larger number of libraries. This approach is successfully utilized by Haskell's monads [29] as well as some more recent extensions (e.g. arrows [47] and applicative functors [42]) and is also used by computation expressions in F# [28]. We find it especially valuable in the situation where there are multiple programming models for a particular domain and the user may want to choose between several options. For this reason, we believe that this is the best way to tackle the reactive programming area as well.

2.4 Note on programming languages

We present the examples in F#, which is a language from the ML family. This language is both practically useful (as it targets the .NET platform and is now included in Visual Studio 2010), but thanks to the ML background, it should be also familiar to the academic community. However, the concepts described in this thesis are directly applicable to many mixed functional/imperative languages including Scala, C# 3.0, and Python, but (somewhat surprisingly) also JavaScript, which is becoming increasingly important.

3. Main contributions of this thesis

The overall goal of the thesis is to develop a reactive framework that makes it possible to combine the declarative and imperative style, easily encode event interactions and to reason about code written using our library. In order to achieve this, we make original contributions to the following three areas:

- **Avoiding space leaks in reactive applications.** When combining the declarative and imperative style of reactive programming, it becomes easy to introduce patterns where the usual garbage collector for objects cannot automatically dispose all components that we intuitively consider garbage.

  We formally define the notion of garbage for reactive applications based on the duality between objects and events and we present a formal algorithm for collection of garbage in this environment.

  Building on top of the theoretical model, we show how to improve the existing F# library [19] so that it doesn't cause leaks when used in the mixed model. This allows us take advantage of the clarity and simplicity of the declarative approach as well as the expressivity of the imperative model.

[2] In Concurrent ML, events are used as the primary synchronization primitive, which is a notion very different to our events, which are used solely as communication channels.

- **Pattern matching for monadic computations.** In order to express waiting for a combination of event occurrences, we design an extension of monads that adds support for pattern matching on monadic values. Our extension keeps not only a familiar syntax of ML pattern matching, but also a familiar semantics, which makes it easy to use it even in a non-standard programming model, but also allows easy reasoning about programs.

  The goal of our design is to create a more widely useful syntactic extension, so we demonstrate it not only using our reactive programming model based on events, but also in several concurrent and parallel programming scenarios. All these three encodings are implemented as a library extension that benefits from the syntax we introduce.

  The extension is implemented in terms of two primitive operations that have to be provided for each type of computation. We analyze these operations formally and describe algebraic laws that should hold about them. We also study how the extension relates to existing types of computations such as commutative monads.

- **Reactive programming library.** Finally, we design a reactive programming library based on the imperative approach. When using the library, the code is structured into state machines, which is a natural way of thinking about complex interaction patterns. The library also makes it possible to write modular reactive code, which makes applications easier to debug. The design also enables easy unit testing, which is otherwise difficult for reactive applications.

  To enable easy reasoning about reactive programs written using the library, we present a formal semantics of the subset of the used language. We also present the notion of semi-discrete time, which is particularly useful for modelling reactive programs where a source event can cause a series of reactions (also events) that all occur before the source event can be triggered again.

3.1 Road map of the thesis

The rest of this thesis is organized as follows. Chapter II reviews the existing work we build on and related work in the field of reactive programming as well as other relevant areas. Chapter III presents a big picture overview of our approach, shows our ultimate goal and reviews the problems that need to be solved.

The next three chapters present the main novel ideas of this thesis. Chapter IV focuses on the problems of memory leaks in event-based reactive applications, Chapter V presents our langauge extension that adds pattern matching to monadic computations and Chapter VI presents our reactive programming model based on the imperative approach and gives a formal semantics of the model.

Finally, Chapter VII discusses several future directions that we'd like to explore and Chapter VIII gives a brief overview of the novel ideas discussed in this thesis that we find the most interesting. The appendices include additional information and proofs of some formal statements presented in the thesis.

Chapter II

Background and related work

This section gives an overview of the existing work that we use as the basis for this thesis. In section 1 we'll briefly review some functional programming techniques that are crucial for our work. In section 2 we'll discuss the existing F# library for *declarative* programming with events. Our imperative library for reactive programming is based on monads which are discussed in section 3. For readers who aren't familiar with the F# language, we provide a brief review of features that are specific to F# in Appendix A.

We'll also discuss several languages and libraries that are related to our work and provided some inspiration. Synchronous languages discussed in section 4 are designed for a different environment (safety-critical embedded systems), but deal with many problems of reactive programming. Concurrent languages that we review in section 5 emphasize synchronization of concurrently executing processes or threads, but contain many useful ideas about expressing complicated interaction patterns. Finally, in section 6, we'll discuss other languages and libraries that deal with reactive programming in the context of application software.

1. Functional techniques

As already discussed, the approach of our thesis is to provide *declarative* and *imperative* programming models for reactive programming. In addition, we provide an extension for expressing operations involving combinations of events based on pattern matching. All of these three techniques are rooted in concepts from functional languages, but lift the standard form of the concept to the reactive environment. In this section, we review the basic functional techniques we build on.

1.1 Pattern matching

This language feature is supported by most of the functional languages. It makes it possible to analyze a value (for example a tuple of lists) and specify what code should be executed for individual cases (for example, when the first element of the first list is smaller than the first element of the second list). This is particularly useful in functional programming, where many data structures are transparent, meaning that we can analyze the underlying values from which the data structure is composed. The following example shows a recursive function `merge` that merges two ordered lists into an ordered list:

```
1: let rec merge xs ys =
2:   match xs, ys with
3:   | x::xs', y::ys' when x <= y -> x::(merge xs' ys)
4:   | x::xs', y::ys' -> y::(merge xs ys')
5:   | rest, [] | [], rest -> rest
6:
7: > merge [ 1; 5; 6; 9 ] [ 1; 2; 3; 7 ];;
8: val it : int list = [1; 1; 2; 3; 5; 6; 7; 9]
```

The pattern matching is written using the `match` construct (line 2) which consists of several clauses (lines 3, 4 and 5). The `merge` function takes two lists of numbers as parameters and it gives a list of tuples as an argument to the `match` construct. We use the `head::tail` pattern to decompose a list into a *head* containing the first element and a *tail* representing the rest of the list.

The first clause deconstructs the tuple using a pattern `first, second` and then uses nested patterns to deconstruct both of the lists into their first heads and tails. When the first element of the first list is smaller than the first element of the second list (line 3), we recursively merge the rest of the first list with the second list and append the first element of the first list to the front of the result. The second clause (line 4) handles the remaining case when both of the lists are non-empty and the last clause (line 5) handle the case when one of the lists is empty.

It is worth noting that a value that matches the first clause (line 3) would also match the second clause (line 4), since the second clause is the same with the exception that it doesn't specify additional condition using the `when` construct. In ML-family of languages, clauses are tested in order in which they are written, which means that the second clause will be called only for values that don't also match the first clause.

Our language extension presented in Chapter V is based on pattern matching, so we find it important to give a comprehensive example. One of the key benefits of our extension is that it preserves the usual syntax and also semantics of ML pattern matching (including the fact that the order of clauses matters).

1.2 Encoding state machines

Another functional technique that inspired our work is the ability to easily encode state machines using recursive functions. The general approach is that each state is encoded as a recursive function and transitions from the state are represented as tail-calls to other functions (note that the use of tail-calls means that the execution of the code in the original state will not continue later). It is worth noting that this is more powerful than usual theoretical *finite-state machines* as functions may keep some additional state in the parameters.

The Figure 2 visualizes a sample state machine that tests whether a numeric sequence consists of ascending sub-sequence followed by a descending sub-sequence. The transitions are annotated with the condition that must hold for the currently processed element *x* and the previous element *prev*.

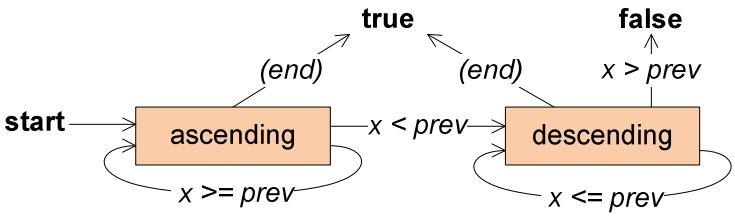


Figure 2. Accepting numeric sequences consisting specific pattern.

According to the previous description, we can encode the state machine using two functions that represent the `ascending` and `descending` state. Our functions also need to keep a state consisting of the previous processed number and the list containing the numbers yet to be processed. The following listing shows the complete implementation including two sample uses of the function:

```
let rec ascending prev l =
  match l with
  | x::xs when x >= prev -> ascending x xs
  | x::xs -> descending x xs
  | [] -> true
and descending prev l =
  match l with
  | [] -> true
  | x::xs when x <= prev -> descending x xs
  | _ -> false
and start l =
  ascending 0 l

> start [1; 3; 4; 1];;
val it : bool = true
> start [1; 3; 1; 3];;
val it : bool = false
```

Many reactive applications follow similar pattern with the exception that transitions are caused by events such as `MouseClick` or combinations of patterns. Code written using our reactive library (presented in Chapter VI) is often structured as state machines similar to the one presented above.

What makes the library interesting is the fact that we often need to have a large number of state machines running in parallel (for example, representing individual features of the application). However, we'd like to implement the system as single-threaded to make sure that it can run correctly on the main GUI (graphical user interface) thread. Since most of the state machines driving the user interface don't perform any extensive computations, the single-threaded model is sufficient and much easier to work with.

1.3 Processing lists with higher-order functions

When working with lists, we can use pattern matching and recursion or we can use higher-order functions that implement frequently used operations for working with lists. These operations can be parameterized and composed into more complex ones, which makes it possible to express complex operations in a simple declarative way.

To give an example, we'll look at the List.map function, which takes a list and a function. It applies the function to all elements of the input list and returns a new list consisting of the values obtained as the result:

```
val List.map : ('a -> 'b) -> list<'a> -> list<'b>

> List.map (fun n -> n + 10) [ 1; 2; 3 ]
val it : list<int> = [ 11; 12; 13 ]
```

The type signature shown on the first line demonstrates that the function is generic and can work with lists containing elements of any type. It also shows that the returned values may be of different type. If we want to apply multiple list processing operations in a sequence, we can use the pipelining operator (`|>`), which passes the value on the left-hand side as an argument to the function on the right-hand side. The following example first filters a list and then calculates the square of all elements of the filtered list:

```
let numbers = [ 1 .. 9 ]
numbers
  |> List.filter (fun n -> n % 2 = 1)
  |> List.map (fun n -> n * n)
```

The important thing to note is that events are in many ways similar to lists. The difference is that list immediately contains all the values, while event generates values one-by-one at different times during the execution of the application. This observation motivates the design of existing *declarative* library for reactive programming which we'll introduce in section 2. The library is also the subject of Chapter IV where we discuss how to improve the existing implementation in order to avoid memory leaks.

2. First-class compositional events

In .NET, an *event* is a special member of a class just like a method or property that can be used only directly. In particular, the only two operations supported by events are registering a handler with an event and unregistering a handler from an event. We would use the first operation in Chapter I in the code that initializes the user interface like this:

```
let btnUp = new Button(Text = "Increment")
let btnDown = new Button(Text = "Decrement")
btnUp.Click.AddHandler(new EventHandler(x.IncrementClick))
btnDown.Click.AddHandler(new EventHandler(x.DecrementClick))
```

This snippet creates two buttons and then registers the methods `IncrementClick` and `DecrementClick` as the handlers of the `Click` events of the two buttons. To do this, we use the `AddHandler` operation and we pass it a newly created value of the `EventHandler` type. This is a delegate type, which can be viewed simply as a wrapper for functions of some specific type that is commonly used to represent handlers of user interface events[3].

[3] There are two reasons why .NET uses delegates instead of simple function values. The first reason is that delegates also provide reference equality comparison and the second reason is that function values aren't directly supported by the .NET runtime.

2.1 Using events as values

The article [19] presents a different approach to working with events, which is used in the F# language. Instead of treating events specially, F# exposes events as normal values implementing some interface that provide ordinary methods for registering and unregistering handlers. This means that in the above example, `btnUp.Click` was just a property returning a value of type `IEvent<EventArgs>`. This is the interface type used to represent events. The two methods provided by the interface are `AddHandler` (for registering handlers) and `RemoveHandlers` (for unregistering handlers).

This may seem like a minor difference, but it enables radically different style of programming with events. We can now write functions that take events as arguments or return newly created events as their result. This makes it possible to define higher-order functions for working with events that are in many ways similar to list processing functions discussed in section 1.3.

The following example demonstrates event handling using this approach. We create a new event that is triggered only when the user clicks using the right mouse button. The value carried by the event will be always 1:

```
val Event.filter : ('a -> bool) -> IEvent<'a> -> IEvent<'a>
val Event.map    : ('a -> 'b)   -> IEvent<'a> -> IEvent<'b>

let rightClicks =
  btnTest.MouseDown
    |> Event.filter (fun e -> e.Button = MouseButtons.Right)
    |> Event.map (fun _ -> 1)
```

The listing first shows the type signatures of the two higher-order functions that we use in the code. Their types are similar to the types of list processing functions from section 1.3, with the difference that instead of lists of type `list<'a>`, we're now working with events of type `IEvent<'a>`. The `MouseDown` property of a button exposes an event of type `IEvent<MouseEventArgs>`, which carries information about the mouse action (such as the mouse button used and coordinates).

To construct the event described above, we first use the `Event.filter` function to construct an event carrying a value of type `MouseEventArgs`, which is triggered only when the click was caused by the right mouse button. Next, we use `Event.map` with a lambda function that ignores the argument and always returns 1 to create the desired event of type `IEvent<int>`.

2.2 Declarative event handling in practice

To demonstrate working with first-class events and higher-order functions for working with them, we'll re-implement the example from section 1. Solutions that are based on composable declarative components are often easy to visualize. The data-flow in our sample application is visualized in Figure 3.

The idea is that we take the click events and turn them into events carrying an integer value just like in the previous example. We'll create events that carry either +1 or –1 depending on which button was clicked. Then we merge these two events and use the `Event.scan` function to sum the values carried by the events.

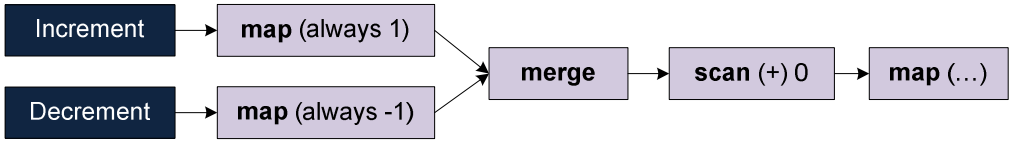


Figure 3. An event-processing pipeline used in the sample application; the two boxes on the left represent source events, and the lighter boxes represent events created using processing functions.

The `Event.scan` function deserves some explanation. It takes an initial state (in our case the number 0) and a function that is used to calculate new state from the original state and the current value carried by the source event. This calculated value is stored as the new value of the state and is also used to trigger the resulting event. The entire behavior can be coded as follows:

```
1: let incEvent = btnUp.Click |> Event.map (fun _ -> 1)
2: let decEvent = btnDown.Click |> Event.map (fun _ -> -1)
3:
4: Observable.merge incEvent decEvent
5:     |> Event.scan (+) 0
6:     |> Event.map (sprintf "Count: %d")
7:     |> Event.add (fun msg -> lbl.Text <- msg)
```

To make the code more readable, we don't encode the whole pipeline as a single expression, even though that would be perfectly possible. Instead, we first declare two helper events (lines 1 and 2). The type of both `incEvent` and `decEvent` is `IEvent<int>`, which means that they represent events carrying integers. The value carried by the event raised by the Increment button is always +1, and the value of the other event is always –1.

We merge these events (line 4) to create an event that will be triggered every time either button is clicked. The event carries integer values, so we can use `Event.scan` to sum the values starting with 0 as an initial value. We're using the plus operator for aggregation, so for each click the function will add +1 or –1. Next, we use `Event.map` to format the carried number as a string (line 6). The last combinator in the processing pipeline registers a handler with the previously created event and runs the specified function whenever the event occurs. It returns a unit value as the result, so it can be used only at the end of the processing pipeline.

2.3 Semantics of declarative event handling

In this section, we'll briefly look at formal definition of the meaning of code written using a subset of the *declarative* event-handling library. We'll provide a similar definition for our *imperative* library later in this thesis, so this can be viewed as an introduction to the problem (although the declarative library hasn't been described formally before). The syntax of our declarative language looks as follows:

evt =	*id*	External event (such as `btn.Click`)
	Event.filter *expr evt*	Filtering event using the specified predicate
|	**Event.map** *expr evt*	Projection using the specified function
|	**Event.scan** $expr_1$ $expr_2$ *evt*	Aggregation using function and initial state
|	**Event.merge** evt_1 evt_2	Merging two specified events

Although the F# library provides several other combinators, the subset presented above covers most of the interesting cases. An event can be constructed from external events (such as events of user interface controls) or by applying some (parameterized) combinator to other event or events. Aside from relatively simple combinators such as `Event.map` and `Event.filter`, we also included the `Event.scan` combinator, which is interesting as it keeps some state and the `Event.merge` which takes multiple events as inputs.

We'll model an event as an ordered set containing time/value pairs. The *value* is anything that can be calculated as a result of evaluating F# expression. We don't define the semantics of F# expressions and we'll simply refer to the definition of Standard ML [54], which is the basis for F#. For now, we'll say that *time* is some abstract value as we won't need to perform any calculations with it (we'll simply use the time provided by external events). The definition of the meaning is given in terms of the two following functions:

$\rightarrow_e \;:\; expr \rightarrow (env \rightarrow value)$
$\rightarrow \;:\; evt \rightarrow (\text{env} \rightarrow [time \times value])$

The first function defines the semantics of a standard expression. When given an environment that contains values of free variables of the expression and the meaning of external events, it produces a value that represents the result. The second function defines the meaning of an event. When given an environment, it returns an ordered set (a list) of time/value pairs as discussed above. The environment can be used as a function, so *env(id)* when *id* is an identifier of an event gives us a list of time/value pairs representing an external event (this is the only use of environments that we'll need in the following definitions). The formal semantics of declarative event processing language is defined by the rules displayed in Figure 4.

$$\frac{e = env(id)}{env \vdash id \rightarrow e} \quad (External)$$

$$\frac{env \vdash evt \rightarrow e \quad env \vdash (expr\ v_i) \rightarrow_e b_i}{env \vdash \textbf{Event.filter}\ expr\ evt \rightarrow \{(t_i, v_i) \in \text{e} \mid b_i = \textbf{true}\}} \quad (Filter)$$

$$\frac{env \vdash evt \rightarrow e \quad env \vdash (expr\ v_i) \rightarrow_e n_i}{\vdash \textbf{Event.map}\ expr\ evt \rightarrow \{(t_i, n_i) \mid (t_i, v_i) \in e\}} \quad (Map)$$

$$\frac{env \vdash evt \rightarrow e \quad env \vdash init \rightarrow_e s_0 \quad env \vdash (expr\ s_{i-1}\ v_i) \rightarrow_e s_i \ \ (\forall i \geq 1)}{env \vdash \textbf{Event.scan}\ expr\ init\ evt \rightarrow \{(t_i, s_i) \mid (t_i, v_i) \in e\}} \quad (Scan)$$

$$\frac{env \vdash evt_1 \rightarrow e_1 \qquad env \vdash evt_2 \rightarrow e_2}{env \vdash \textbf{Event.merge}\ evt_1\ evt_2 \rightarrow e_1 \cup e_2} \quad (Merge)$$

Figure 4. Formal semantics of declarative event processing.

The above definition describes a model without side-effects, meaning that a function provided, for example, as a predicate cannot print to the console or perform any other side-effect. However, in practice this is possible in F# and so it is reasonable to discuss when the function will be evaluated. For example, will we listen to an event even if there are no handlers attached to it? We'll focus on this question in more details in Chapter IV.

We'll conclude this section with a brief overview of some limitations of this model:

- **Events never end.** The events described above never end, which limits the expressive power of the model. For example, we cannot write a combinator that would behave as one event until the event ends and then ran a function to get another event and continue behaving as the second event.
- **Limited expressivity.** It is worth noting that we cannot write recursive declarations of events. This has positive consequences, most notably the fact that processing an event cannot diverge (if we use predicates and functions that don't diverge). On the other hand, this clearly shows that the expressivity of declarative processing using is very limited. In Chapter IV, we'll also look at example where encoding relatively simple behavior is surprisingly difficult.
- **Encoding state machines.** The declarative approach based on the combinators presented above is also not powerful enough to provide some natural way of encoding state machines. This is one of the problems that motivated the work in this thesis (especially in Chapter VI).
- **Removing event handlers.** The implementation of event combinators presented in [19] and discussed in this section doesn't support removal of registered handlers. This means that if we create an event value and then use it imperatively (using `AddHandler` and `RemoveHandler`) we may introduce memory leaks. We solve this limitation in Chapter IV.

In this section, we introduced first-class compositional events that are available in F# and that are used (in an improved form) as one part of our reactive programming technique. The second part that we'll present later is based on monads and on F# computation expressions that are introduced in the next section.

3. Monads and computation expressions

Monads are a theoretical concept coming from category theory, which has proven useful in computer science [59]. Monads can be viewed as a useful abstraction for representing some types of computations [29] and some languages even provide a convenient notation for writing monadic computations [60]. In this section, we'll give a brief overview of monadic computations. Next, we'll look at language extension for writing monadic computations provided by F# and finally, we'll look how to encode monads and related computation types using this language feature.

3.1 Monadic computations

A monadic computation is a computation that has some special aspect or effect. In pure languages like Haskell, monads can be used for creating computations that maintain and modify some local state or perform side-effects such as printing to the console. This is possible, because monadic computations can be composed in a

way that makes the computation sequential and as a result, the side-effects will be run in a well-defined order.

However, monadic computations can be also used to represent computations that have some non-standard aspect. For example, the `Maybe` monad represents computations that may fail at any time returning a special value `Nothing` as the result. The `List` monad can be used to write non-deterministic computations that can yield multiple results. The most prominent example of monadic computation in F# is called *asynchronous workflows* and represents a computation that may produce result at some later time (by calling a function specified by the user of the workflow).

Standard definition. A monadic computation is represented by some generic type `M<'a>` where the type parameter specifies the type of value (or values) produced as the result of monadic computation (internally, the type may be for example a function or list). When writing code using monadic computations, we don't use the underlying type directly and instead, we use two operations that every monadic computation must provide. The operations define the behavior of the monad and have the following type signatures (for some monad `M<'a>`):

```
bind   : M<'a> -> ('a -> M<'b>) -> M<'b>
return : 'a -> M<'a>
```

In order to define a correct monad, the two operations need to satisfy certain algebraic equations. These equations guarantee that we can rewrite a monadic expression in certain ways without changing its meaning and are also useful for proving properties about monadic computation in theory. The following definition shows the three *monad laws* described in [29]. We denote the `bind` operation as "≫=":

$$\text{return } a \gg\!= f \;\equiv\; f\,a \qquad \text{(Left identity)}$$
$$m \gg\!= \text{return} \;\equiv\; m \qquad \text{(Right identity)}$$
$$(m \gg\!= f) \gg\!= g \;\equiv\; m \gg\!= (\backslash x \rightarrow f\,x \gg\!= g) \qquad \text{(Associativity)}$$

Alternative definition. The previous formulation is useful when working with monads in practice, because the `bind` operation represents sequential composition of monadic computations (and so it is suitable for encoding of computations). In some cases, it is more convenient to use an alternative definition which is more closely related to the original concept from category theory. This definition is equivalent and uses the following three operations:

```
map  : M<'a> -> ('a -> 'b) -> M<'b>
join : M<M<'a>> -> M<'a>
unit : 'a -> M<'a>
```

To show that the two definitions are, indeed, equivalent, we need to show how to implement these three functions in terms of two operations presented above and vice-versa. The following listing shows the implementation:

```
let join a  = bind (fun v -> v) a
let map f a = bind (return << f) a
let unit v  = return v
```

```
let bind a f = join (map f a)
let return v = unit v
```

In addition to implementing the new set of functions in terms of the first one, we also need to re-formulate the three monad laws. This can be done simply using the implementation, but we can get a more elegant formalization as presented in [45]. In this thesis, we'll mostly work with the first definition of monad (which is more usual from the programming point of view), however Chapter V uses the second definition in several places as it makes the discussion easier to follow.

3.2 Asynchronous workflows

Perhaps the most prominent example of monadic computation in F# is called *asynchronous workflows* [4]. It represents potentially long-running computations such as I/O that may be executed in background by the system. Asynchronous workflow is started by giving it a callback that should be called when the workflow completes. It may run eagerly and trigger the callback when it completes, but more often it uses some system call that takes a callback as an argument (such as asynchronous socket operations). This means that asynchronous workflows allow us to write complex I/O operations without blocking a system thread.

In F#, the two monadic operations are combined in a class type (discussed in Appendix A) with members `Bind` and `Return`. The library provides a single instance of the class called *computation builder* that is used when writing code using the computation expression syntax. The following example uses the `async` builder to write a workflow that downloads the content of a web page:

```
let downloadUrl(url) = async {
   let req = HttpWebRequest.Create(url)
   let! rsp = req.AsyncGetResponse()
   let rd = new StreamReader(rsp.GetResponseStream())
   let! html = rd.AsyncReadToEnd()
   return html }
```

The body of the function is enclosed in the `async { … }` block which denotes that we're creating a computation expression defined using the `async` builder. Inside the block, we can use some non-standard constructs that are translated to calls to the computation builder. In this example, we're using `let!` which corresponds to `bind` and `return`, which corresponds to the monadic `return` operation. For example, the value returned by `AsyncReadToEnd` has a type `Async<string>`, but we're using monadic bind, so the type of the `html` value is `string`. When the F# compiler processes the computation expression, it transforms the computation to the following code:

```
1: let downloadUrl(url) =
2:    let req = HttpWebRequest.Create(url)
3:    async.Bind(req.AsyncGetResponse(), fun rsp ->
4:       let rd = new StreamReader(rsp.GetResponseStream())
5:       async.Bind(rd.AsyncReadToEnd(), fun html ->
6:          async.Return(html)))
```

The two operations that return value of type `Async<'a>` are primitive asynchronous workflows provided by the F# library that perform some long running operation that can be implemented by providing callback to the system. For example, on the line 3, we use `Bind` to compose a workflow returned by `AsyncGetResponse` with the rest of the computation, which is provided as a lambda function. The `Bind` operation decides when to run the function. It may run it after the server replies, but if the network communication fails, the function may not be executed at all.

The value returned from the `downloadUrl` function is the result of the `Bind` operation. In our example, it will be a value of type `Async<string>`. This shows that computation expressions provide a way for *composing* computations. We have some basic workflows and we can build more complicated ones that represent more complex operations. To run the returned workflow, we need some operation that understands the underlying structure of the type and knows how to run it. In case of asynchronous workflows, there are several functions such as `Async.Spawn`, which starts the operation on a background thread.

3.3 Computation expressions

In this section, we'll briefly review the most important parts of the F# computation expression syntax and we'll look how they are translated to functions provided by the computation builder. The F# language specification [30] describes the syntax and translation in full details. We'll concern ourselves with the following subset of the full syntax:

expr	= *expr* { *cexpr* }	Computation expression
cexpr	= **let** *pat* = *expr* **in** *cexpr*	Binding value
	| **let!** *pat* = *expr* **in** *cexp*	Binding computation
	| **return** *expr*	Return result
	| **return!** *expr*	Return computation
	| **yield** *expr*	Yielding value
	| **yield!** *expr*	Yielding computation
	| $cexpr_1$ **;** $cexpr_2$	Composing computations
	| **if** *expr* **then** $cexpr_1$ **else** $cexpr_2$	Conditional computation
	| **if** *expr* **then** *cexpr*	Conditional computation
	| **match** *expr-list* **with** $pat\text{-}list_i$ -> $cexpr_i$	Value pattern matching

The syntax includes several constructs that are not available in the core F# language and that are directly translated to calls to computation builder operations. This includes the constructs we already introduced (`let!` and `return`) and `return!` Constructs `yield` and `yield!` are in many ways similar to `return`/`return!` pair. They are mainly useful when we need to syntactically distinguish a computation that may produce multiple values, because the `yield` keyword seems more natural in this case. We'll see an example of such computation in the next section.

The syntax also provides variants of several standard expressions such as `if`, `match` and sequencing using semicolon. Some of them can be used without providing any special support in the computation expression builder, while some of them require a special support. Similarly to these, the syntax also allows using `for` and `while` loops as well as resource management constructs `use` and `use!`, and constructs for exception handling which are all discussed in [30].

The F# compiler performs a simple syntactical transformation, which means that if we don't provide some of the operations (e.g. to support `yield`), it will not give any error unless the user writes `yield` in the computation. This makes the design very flexible as we may provide only some of the primitives to create different kinds of computations. The following list shows all the operations that a computation builder may provide to support the syntax presented above:

```
type MonadBuilder =
  abstract Bind    : M<'a> -> ('a -> M<'b>) -> M<'b>
  abstract Return  : 'a -> M<'a>
  abstract Yield   : 'a -> M<'a>
  abstract Combine : M<'a> -> M<'a> -> M<'a>
  abstract Zero    : M<unit>
```

Computation expressions don't specify any meaning of these operations and even the type signature isn't technically enforced. However, a computation should obey some laws (such as the monad laws discussed earlier). We'll describe the translation that transforms a code written inside `m { … }` block to an ordinary F# code using the following function:

$$[\![-]\!]_{\text{cexpr}} : \mathit{cexpr} \rightarrow \mathit{ident} \rightarrow \mathit{expr}$$

The function takes a construct from the syntactic category of computation expressions and an identifier (which refers to the computation builder) and produces a value from the syntactic category of ordinary F# expressions. The translation of computation expressions is started as follows:

$$[\![\mathit{expr}\ \{\ \mathit{cexpr}\ \}\ ]\!] = \textbf{let}\ \mathrm{m} = \mathit{expr}\ \textbf{in}\ [\![\mathit{cexpr}]\!]_{\text{cexpr}}\ \mathrm{m}$$

Note that the construct preceding the block is can be any F# expression. The compiler assigns the result of this expression to a new variable to avoid evaluating its side-effects multiple times and then uses the identifier. The translation rules that define the function are shown in **Figure 5**.

$$\begin{aligned}
[\![\textbf{let}\ \mathit{pat} = \mathit{expr}\ \textbf{in}\ \mathit{cexpr}]\!]_{\text{cexpr}}\ m &= \textbf{let}\ \mathit{pat} = \mathit{expr}\ \textbf{in}\ [\![\mathit{cexpr}]\!]_{\text{cexpr}}\ m \\
[\![\textbf{let!}\ \mathit{pat} = \mathit{expr}\ \textbf{in}\ \mathit{cexpr}]\!]_{\text{cexpr}}\ m &= \mathrm{bind}_{\mathrm{m}}\ (\textbf{fun}\ \mathit{pat}\ \text{->}\ [\![\mathit{cexpr}]\!]_{\text{cexpr}}\ m)\ \mathit{expr} \\
[\![\textbf{return}\ \mathit{expr}]\!]_{\text{cexpr}}\ m &= \mathrm{return}_{\mathrm{m}}\ \mathit{expr} \\
[\![\textbf{return!}\ \mathit{expr}]\!]_{\text{cexpr}}\ m &= \mathit{expr} \\
[\![\textbf{yield}\ \mathit{expr}]\!]_{\text{cexpr}}\ m &= \mathrm{yield}_{\mathrm{m}}\ \mathit{expr} \\
[\![\textbf{yield!}\ \mathit{expr}]\!]_{\text{cexpr}}\ m &= \mathit{expr} \\
[\![\mathit{cexpr}_1\ \textbf{;}\ \mathit{cexpr}_2]\!]_{\text{cexpr}}\ m &= \mathrm{combine}_m\ \mathit{cexpr}_1\ \mathit{cexpr}_2 \\
[\![\mathit{cexpr}\ \textbf{;}\ \mathit{cexpr}]\!]_{\text{cexpr}}\ m &= \mathrm{combine}_m\ \mathit{cexpr}\ \mathit{cexpr} \\
[\![\textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{cexpr}]\!]_{\text{cexpr}}\ m &= \textbf{if}\ \mathit{expr}\ \textbf{then}\ [\![\mathit{cexpr}_1]\!]_{\text{cexpr}}\ m\ \textbf{else}\ \mathrm{zero}_{\mathrm{m}} \\
[\![\textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{cexpr}_1 & \\
\quad \textbf{else}\ \mathit{cexpr}_2]\!]_{\text{cexpr}}\ m & = \begin{array}{l} \textbf{if}\ \mathit{expr}\ \textbf{then}\ [\![\mathit{cexpr}_1]\!]_{\text{cexpr}}\ m \\ \quad \textbf{else}\ [\![\mathit{cexpr}_2]\!]_{\text{cexpr}}\ m \end{array} \\
[\![\textbf{match}\ \mathit{expr\text{-}list}\ \textbf{with} & \\
\quad \mathit{pat\text{-}list}_i\ \text{->}\ \mathit{cexpr}_i]\!]_{\text{cexpr}}\ m & = \begin{array}{l} \textbf{match}\ \mathit{expr\text{-}list}\ \textbf{with} \\ \mathit{pat\text{-}list}_i\ \text{->}\ [\![\mathit{cexpr}_i]\!]_{\text{cexpr}}\ m \end{array}
\end{aligned}$$

Figure 5. Translation rules for F# computation expressions

It is worth noting that, unlike Haskell, the F# type system doesn't support higher kinded types, so it isn't possible to write code generic over the monad type. In the usual F# use, this doesn't seem to be a frequent problem.

As already noted, the F# compiler doesn't place any restrictions on which operations should the computation builder support. However, the usual definitions follow several patterns that are based on certain algebraic structures that are useful for representing computations. We'll review the common structures in the next section.

3.4 Monads, monoids, additive monads

Computation expressions can be used to encode three common structures (or "design patterns") used in functional programming. The first two of them are related to well-known mathematical objects from algebra and the last one is a combination of the first two types. F# computation expressions are interesting by their ability to express all three patterns using just a single language feature.

Monoids. Mathematically, a monoid is a set S together with an associative binary operation $\cdot$ and an identity element $i \in S$ such that $\forall a \in S: i \cdot a = a \cdot i = a$. An example of monoids known from mathematics is the set of natural numbers with zero as the identity element and addition as the operation or a set of natural numbers with 1 and multiplication.

However, monoids also appear in many scenarios in programming. There are many data structures that can be viewed as monoids. For example a string together with an empty string as the identity and concatenation forms a monoid. A similar example is a list (with empty list and concatenation).

Let's now look how we can use F# computation expressions for working with the string monoid. When declaring a computation builder for working with monoids, we need to provide the `Zero` member (which will return the identity element) and the `Combine` member which will apply the binary operation to the two arguments. In addition, we also need to provide `Yield` element, so that the user has some way for passing individual elements from the computation to the monoid:

```
type StringMonoid() =
   member x.Combine(s1, s2) = String.Concat(s1, s2)
   member x.Zero() = ""
   member x.Yield(s) = s

let str = new StringMonoid()
```

Now we can use this computation to write code that generates a string value as the result. The simplest possible example would look like this:

```
> str { yield "Hello "
        yield "world!" };;
val it:string = "Hello world"
```

The F# compiler translates the computation above to a call to the `Combine` operation with two calls to the `Yield` primitive as arguments. However, the key interesting thing about computation expressions is that we can use normal F# code inside the computation as well and the *monoid computation* is used only to accumulate the result. The following function shows an example that creates a string with nicely formatted numbers in a specified range:

```
1: let rec numbers a b = str {
2:    yield string a
3:    if a < b then
4:       yield ", "
5:       yield! numbers (a + 1) b }
6:
7: > numbers 1 3;;
8: val it : string = "1, 2, 3"
```

The code is implemented as a recursive function. It starts by returning the first number in the range formatted as a string (line 2). Next, it checks whether it needs to generate more numbers. If yes, it adds a separator to the result (line 4) and then recursively generates all numbers within a range starting from the next number. Note that we're using `if .. then ..` expression without the `else` branch. In this case, the F# compiler automatically inserts a call to the `Zero` member to the `else` branch, so the last iteration will compose the number with the identity element, which is an empty string.

Monads. We already presented an example of computation expression used for encoding monads when talking about asynchronous workflows, because asynchronous workflows are based on the Continuation monad. Other useful examples of monads include a computation that may fail at any point, a computation that has read-only access to some state, a computation that has read/write access to some state or for example a computation that can be executed step-by-step (that is, the computation is automatically divided into steps and we can run a single step, then perform some other work and then run the next step).

To demonstrate working with monads in F#, we'll create a computation builder for working with the monad that may fail. We'll use the option<'a> type as the monadic type. This is a discriminated union with two cases. A failure is represented using the `None` case and a success that produced a value is represented as `Some(value)`. To define a monadic computation, we need to provide the members `Bind` and `Return`, which can be done as follows:

```
type MaybeMonad() =
   member x.Bind(m, f) =
      match m with
      | Some(v) -> f v
      | None -> None
  member x.Return(v) = Some(v)

let maybe = new MaybeMonad()
```

The `Return` operation simply wraps the *value* into a *monadic value* that represents a computation that succeeded. The Bind operation is more interesting. It gets a value of type `option<'a>` as the first argument and a function that takes a value `'a` and produces the result of type `option<'b>` as the second argument. If the first argument represents a computation that failed, we also return failure (because there is no way of obtaining a value of type `'a`). If the computation contains a value, we extract the value and run the rest of the computation (passed as a function).

To demonstrate the computation, let's say that we have a function `tryReadNumber` that reads a string from the user and tries to convert it to an integer. If it fails, it returns `None`. We also have a function `db.TryFindProduct`, which takes an ID of a product and tries to find it, returning `None` if the product doesn't exist. Then we can write a function that reads an ID from the user and tries to find the name of the specified product like this:

```
1: let rec productNameByID() = maybe {
2:     let! id = tryReadNumber()
3:     let! prod = db.TryFindProduct(id)
4:     return prod.Name }
```

When performing an operation that may fail, we need to call it using the let! construct from our computation. This way we can access the actual value in case the computation succeeds. If the computation fails when reading a number from the user or when searching for a product, it will return `None` automatically. The return operation at the end of the computation expression wraps the name of the product into a value of type `option<string>` containing `Some(name)` which is returned when the computation succeeds.

Additive monads. The last type of computations that we'll discuss combines the features of *monoids* and *monads*. In Haskell, it is represented by the `MonadPlus` type class. It allows us to produce multiple elements and to compose them using a provided binary operation (just like *monoid*), but it also allows composing computations using the `let!` construct.

Perhaps the most prominent example of this type of computations is a type `list<'a>`. To define a monoid, we'll provide the `Zero` member (returning an empty list), the `Yield` member (which creates a singleton list) and the `Combine` member (which concatenates the lists). Note that the `yield` construct plays the same role as the `return` construct for monads, so we can choose which construct to use depending on which syntax seems more appropriate. To make the computation a *monad*, we also need to provide the `Bind` opertion, which can run the rest of the computation for all values available in the given list and then concatenate all the produced lists. The following computation builder gives the definition:

```
type ListMonadPlus() =
  member x.Zero() = []
  member x.Yield(v) = [v]
  member x.Combine(a, b) = a @ b
  member x.Bind(l, f) = l |> List.map f |> List.concat

let list = new ListMonadPlus()
```

The implementation of the computation builder is mostly straightforward. The only interesting operation is Bind which first uses `List.map` to create a list of lists and then uses the `List.concat` function (corresponding to monadic *join*) to flatten the list. We can use this computation for example to generate a list of cities in a rather sophisticated fashion:

```
> let cities = list {
    yield "York"
    yield "Orleans" }

  let moreCities = list {
    let! n = cities
    yield n
    yield "New " + n }
  ;;
val moreCities : list<string> =
  [ "York"; "New York"; "Orleans"; "New Orleans" ]
```

The first declaration uses only the monoid part of the computation and it creates a list containing two cities. The second computation uses `let!` to run the rest of the computation (which returns the name and the name prefixed with "New") for both of the cities. After the concatenation, this gives us a list containing 4 city names.

4. Synchronous languages

Although our work focuses on the development of standard computer applications, we can find useful inspiration in languages that were developed for safety-critical embedded systems such as flight control systems. The programming model used in synchronous languages [6] has some interesting properties:

- These languages are based on solid mathematical foundations that make it possible to formally reason about programs. Although this isn't critical for our use, we try to keep the model formally tameable and present several examples of formal reasoning as well.
- They are designed for simple execution model, where a clock tick or an event causes a reaction that takes a finite time and memory. When developing embedded systems, we need to prove this property formally. This aspect isn't critical for our work, but it raises some interesting future possibilities.

Out programming model is in many ways similar to the one used by synchronous languages and the solution we present bears similarity to two languages from the synchronous world. In this section, we'll give a brief overview of the declarative language Lustre and imperative language Esterel.

4.1 Declarative Lustre

The Lustre language [10] is designed for sample-driven architectures (with a clock). To make it possible to reason about programs easily, it requires that programs contain no zero-delay loops, which means that a calculation may not depend on its result from the current time tick and as a result, it cannot hang.

In Lustre, a variable represents a time-varying value. This means that we're writing equations that define the output of some computation for all clock ticks at once as opposed to specifying just a single value for the current time. For example, the expression `input + 1` means that for each tick, the output value will be the input value at that tick incremented by one (a constant can be viewed as a constant stream). In addition, Lustre provides two operators that have more complex behavior with respect to time:

- **Previous.** The expression `pre(stream)` gives us the value of `stream` variable in the previous clock tick. At the time 0, the value is undefined.
- **Initialization.** To avoid using undefined values, we can write `init -> stream`, which uses the value of `init` at time 0 and then the value of `stream`.

The following example, adapted from [6], shows a simple Lustre program that counts the number of rising edges in a stream `input`:

```
edge = false -> (input and not pre(input));
edgecount = 0 -> if edge then pre(edgecount) + 1
                         else pre(edgecount);
```

The first definition specifies that the value of `edge` is `true` only when the value of `input` is presently `true` and has been `false` in the previous step. This means that the result will be `true` only at the single tick when the value of `input` changes from `false` to `true`. The second definition specifies that the value of `edgecount` is incre-

mented by one at the tick when the value of `edge` is `true` and stays the same in all other ticks. It is worth noting that whenever we use the `pre` operator inside a definition, we also use the `->` operator to provide the initial value, because the first step would be otherwise undefined.

Lustre programs can be structured using nodes, which are functions that take time-varying values as arguments and return them as the result. Types such as `int` or `bool` are interpreted as streams of `int` and `bool` values, respectively, that change with every clock tick. The following node implements a resettable counter that counts the number of times `input` was set to `true` and can be restarted by setting the value of `reset` stream to `true`:

```
1: node COUNT(input, reset : bool)
2:   returns (count : int);
3: let
4:   count = if (true -> reset)
5:             then 0
6:             else if input then pre(count) + 1
7:             else pre(count);
8: tel
```

The node takes two streams of type `bool` as parameters and returns a single stream of type `int`. At the first tick or when the value of `reset` is `true`, the value of `count` is set to zero (lines 4 and 5). Otherwise, we increment the previous value when the value of `input` is `true` or keep the previous value of `count`.

Now that we've explored a small example of working with Lustre in practice, let's review some of its properties. We already claimed that Lustre uses declarative programming model similar to the one used when writing declarative event handling code using higher-order functions in F# (discussed in section 2.2):

- A difference is that F# events happen one at the time, while Lustre signals are processed all at once when a clock tick occurs. This is a fundamental difference between the programming models. However, we could simulate the Lustre behavior to some extent by having a global tick event and synchronizing all external events with this global clock (this might be appropriate for example for an action game that is implemented using a busy loop).
- The synchronization of streams in Lustre makes it possible to implement operators that take two streams as arguments and perform some calculation with the current value (e.g. `inputA + inputB`). In F#, one event may occur multiple times before the second one and so it is a question how many times should the resulting event be triggered. However, operations that take a single stream such as `input + 1` can be implemented using `Event.map`:

  ```
  input |> Event.map ((+) 1)
  ```

- In Lustre, *nodes* are functions that take time-varying values as arguments and return them as results. We can structure declarative event-driven programs in F# similarly by writing functions that take `IEvent<'a>` values as parameters. A special (and more expressive) case of these functions are higher-order functions for working with events that also take a function as argument. As far as we're aware, this construct isn't supported in Lustre, but could be probably easily provided using code inlining.

- Lustre programs may not contain syntactically recursive definitions that refer to a value of a stream inside its definition without using the `pre` operator. The functions for working with events discussed in section 2.3 don't provide any way for creating recursive references either[4]. If we wanted to access the previous value of an event, we could use the `Event.scan` function.

The programming model developed in this thesis combines declarative and imperative approach. The Lustre language is similar to the declarative model and incidentally, there is another synchronous language, which is similar to our imperative programming model.

4.2 Imperative Esterel

Programs in the Esterel language [11, 12] are imperative and are written as a sequence of imperative constructs such as assignments and loops. A program can consist of multiple threads that execute concurrently and communicate via signals. Similarly to the Lustre language, Esterel is synchronous and uses a single global clock. When the clock ticks, each thread may perform one step of the execution.

Within a single step, the thread may, for example, set values of some signals or mutate some state. However, each step must end with some construct where the thread starts waiting for some event or the next tick. In addition, a step must finish in a finite time and the Esterel compiler uses theorem proving techniques to guarantee that this property holds.

We demonstrate the Esterel langauge using program that implements a behavior similar to the one we've just seen in Lustre. The following program adapted from [11] implements a single thread that counts the number of times `CLICK` signal occurred and can be reset by setting the `RST` signal. The count is reported using the `VAL` signal:

```
1: module Counter:
2: input RST, CLICK;
3: output VAL(integer);
4:
5: var v : integer in
6:   do
7:     v := 0;
8:     loop
9:       await CLICK;
A:       v := v+1;
B:       emit VAL(v)
C:     end
D:   watching RST;
E: end
```

The module declares that it takes two input signals that don't carry any value (at each tick, they are either set or not) and a single output signal that carries an integer (lines 2 and 3). The body declares a local integer value to store the count (line 5). Note that unlike in Lustre, this is an ordinary integer value that we will mutate at various times as the thread executes. The body is implemented using the `do` ...

[4] However, it is possible to create a recursive event declaration using the underlying type that represents events. This may give a valid definition or a construction that causes infinite looping.

`watching` statements, which keeps the body running until the `RST` signal is set. When that happens, the thread stops, so we'd need an additional loop to implement the behavior in the previous example. Inside the body, we first initialize the local variable to 0 (line 7) and then run a loop.

When the thread is started, it will reach the loop within a single clock tick. As noted earlier, the execution of a step must finish in a limited time. To achieve this, we start waiting for the `CLICK` signal (line 9). This operation waits until the first next tick when the signal is set, which prevents us from creating an infinite loop. When the signal is set, we increment the local variable, emit the current count through the `VAL` signal and then continue looping (all within a single time step). The Esterel language is closely related to the imperative programming model that we develop and present in Chapter VI:

- Our imperative model is very similarly structured. Programs are written as computations that consist of imperative statements, although we prefer to use recursion for looping. Our programming model also has a construct corresponding to the `emit` and `await` from Esterel.
- When reacting to an event, an Esterel program needs to finish in a finite time, usually by waiting for another event or the next tick. Our programming model doesn't require this property to be proven formally, but a program needs to start waiting after performing some response in order to be well-behaved.
- Threads in Esterel are actually executing concurrently (using different components of the embedded system). Our imperative model doesn't contain any global clock and concurrent execution in this scenario is difficult. As the result our model is single-threaded, which requires, for example, different treatment of time.

Lustre and Esterel are in many ways similar to our declarative and imperative programming models, however, to our best knowledge, they cannot be combined when developing a single system.

5. Concurrent languages

Many high-level languages for concurrent programming use sophisticated constructs for expressing synchronization between concurrently executing threads. In our reactive programming model, we don't need to synchronize threads, but we need to encode patterns for synchronization of events. That is, we want to express that a composed event should be triggered when the underlying events are triggered according to some pattern and possibly also carrying some values.

In this section, we'll look at some interesting synchronization primitives. The languages based on join calculus provide synchronization primitive called *join* or *chord*, which can be used for encoding rendezvous, but can also encode data transfer that blocks only one or none of the involved threads (which is especially useful for asynchronous programming).

The Concurrent ML language is based on the idea of events, which differs from our notion, because it synchronizes the sender and the receiver of the event (and as such can have only one receiver). However, it provides some interesting ideas

for working with events. Finally, we'll also look at the Manticore language, which is an extension of Concurrent ML that provides syntactic support for many frequent parallel programming patterns.

5.1 Join calculus languages

Join calculus is a formalism [31], which provides a foundation for distributed programming languages. However, it has also proven useful as an inspiration for the design of programming languages dealing with concurrency. We'll briefly review two programming languages that are based on Join calculus. Cω adds concurrency abstractions to the C# language and JoCaml extends the functional language OCaml (which in many ways influenced F#).

Cω language. Cω [33] extends C# by adding two new concepts: *asynchronous methods* and *chords*. An asynchronous method is a method that can be called without blocking the caller and that are not guaranteed to complete immediately after they are called. A chord is a synchronization pattern that includes multiple method declarations separated using '&' followed by a single body. The body of the chord runs only when all of the methods that form the chord have been called.

Method calls are implicitly queued by the runtime. When we call an asynchronous method that cannot run immediately (because it is declared as part of some chord that requires other method calls) is queued, but the caller can continue immediately without blocking. However, when calling a synchronous method that returns some value, the caller blocks until the method call is dequeued and a body of some chord is executed (and a value is returned).

The following example (adapted from [33]) implements a concurrent unbounded buffer. The buffer is represented as a class with an asynchronous method `Put` that takes a string and a synchronous method `Get` that returns a string:

```
public class Buffer {
  public string Get() & public async Put(string str) {
    return str;
  }
}
```

After creating an instance of the `Buffer` class, the user can call both `Get` and `Put` methods. When calling `Put` from some thread, the method call will be queued (together with the string value passed as the argument to `Put`) without blocking the thread. When calling the `Get` method, there are two possibilities. When there is a pending call to the `Put` method, the call will be dequeued and the body of the chord will run immediately. As a result the argument that was originally given to the `Put` method will be returned as the result to the caller of `Get` method. On the other hand, if there are no queued calls to the `Put` method, the caller will be blocked until a value is provided by a call to `Put` by some other thread.

JoCaml. The JoCaml language [32] has been developed prior to Cω with the main focus on distributed systems. It doesn't distinguish between synchronous and asynchronous method calls, so when we need to return a result from a *chord* (named *join* in JoCaml), we need to pass a continuation to one of the methods.

The following example implements a more sophisticated buffer using JoCaml. In the previous implementation, we didn't have access to the entire queue of strings, so it was for example, impossible to provide method returning the number of elements in the buffer. The following implementation doesn't rely on the implicit queuing and stores the elements in a list:

```
1: def push(v) & Empty() = Some(v)
2:  or push(x) & Some(xs) = Some(x::xs)
3:  or pop(r) & Some(xs) =
4:    match xs with
5:    | x::[] -> r(x) & Empty()
6:    | x::xs -> r(x) & Some(xs)
```

The implementation provides two public functions called `push` and `pop` and two private functions `Some` and `Empty` that are used to store the state of the stack. We need to distinguish between the case when the stack is empty and when it contains some value, because we can only accept a call to the `pop` method when there is a value in the stack.

This case of non-empty stack is handed in the join on line 3. When there is only a single element in the stack (line 5) we return it through the continuation and set the state of the stack to `Empty`. When there are more values (line 6) we return the first one and indicate that the stack still contains some values by calling `Some`. Note that when a user calls the `pop` method, but the state of the stack is `Empty`, the call will be blocked until a call to the `push` function is made (from another thread), which in turn invokes the function `Some` (line 1 or 2).

Pattern matching for Joins. The previous example was somewhat cumbersome. We needed to define two local helper methods, because there is no way to write a join that would fire only when the `Some` method was called with a non-empty list as the argument. This motivated the extension [51], which adds support for writing ML-style patterns (as discussed in section 1.2) in the declaration of functions that form the join.

The following example implements the same stack as the one shown in the previous example, but uses patterns for encoding join that can be triggered only when there is a call to `pop` and the stack contains a non-empty list. In fact, the extended JoCaml compiler translates the following code to the one we presented in the previous section.

```
def pop(r) & State(x::xs) = r(x) & State(xs)
 or push(x) & State(xs) = State(x::xs)
```

The first join essentially corresponds to the last join from the previous example. It can run only when there is a pending call to `pop` and a call to `State` with a non-empty list as the argument. When the `State` function has been called previously with an empty list, we need to wait until a call to `push` function is made and an element is added to the stack.

The existing work on languages based on Join calculus is related to this thesis in two ways. Firstly, we take some of the ideas used in these languages and take them over to the reactive scenario and secondly, we introduce a general-purpose

language extension for F# that is capable of encoding a programming model based on the Join calculus:

- Our reactive programming model provides a construct that encodes waiting for an event (similarly to `await` from Esterel). It is possible to wait for multiple events in a way that is similar to waiting for multiple pending calls in the Join calculus. Moreover, we also support pattern matching on the values carried by events, which gives the language additional expressive power, similar to the extended variant of JoCaml.
- As we can see, the embedding of Join calculus in JoCaml is based on a syntax similar to pattern matching and can be extended to support pattern matching to a larger extent. Our extension of the F# language with monadic pattern matching (Chapter V) has a similar expressive power, but at the same time, it supports programming models other than Join calculus.
- The discussion in [51] also presents an interesting observation about the `when` clause which can be used in ML-style pattern-matching constructs (demonstrated in section 1.1). The authors note that while supporting patterns inside declarations of individual functions can be done by a simple translation to the restricted language, supporting the `when` clause is difficult and would largely affect the performance. This observation also applies to some of the programming models that can be implemented using our F# extension.

5.2 Concurrent ML and Manticore

Concurrent ML [57] is a concurrent library implemented on top of the SML language (a language from the ML-family). Concurrent ML programs are structured into multiple threads that can communicate and synchronize by sending messages via channels and receiving them. This makes it possible to implement a wide range of concurrency primitives using this, relatively simple, library.

The syntax of Concurrent ML is the same as the syntax of SML and all concurrent programming features are provided as functions. The Manticore language [38] builds on top of Concurrent ML and adds syntactic support for several common patterns used in parallel programming, such as the support for futures.

Concurrent ML. As already mentioned, concurrently executing threads in Concurrent ML (CML) can communicate via channels. The communication is based on rendezvous. Obviously, when receiving a message from a channel, the receiver is blocked until some sender provides a message, but a similar mechanism applies to the sender as well. When sending a message, the sender is blocked until some other thread receives the message.

To implement this communication, Concurrent ML provides two blocking functions `send` and `recv` that take a channel and send or receive a value. The novel feature of CML is that it decouples sending or receiving of a message and a synchronization that is performed afterwards. The value that represents synchronization is called *event*. When reading a value from a channel, we can use the blocking `recv` function, but CML also provides a non-blocking function `recEvt` that returns an event on which we can synchronize at some later point. To synchronize on an event, we can use the (blocking) `sync` function:

```
val recEvt : channel<'a> -> event<'a>
val sync : event<'a> -> 'a
```

The blocking function `recv` is simply implemented as a composition of `recEvt` and `sync`. However, the most interesting aspect of Concurrent ML is that it provides numerous combinators for creating events that synchronize in a more sophisticated way. We can for example construct an event that synchronizes on the first possible event from a list. Some of the primitive functions for composing events are shown in the following listing:

```
// Creates event representing non-deterministic choice of events
val choose : list<event<'a>> -> event<'a>

// Creates a new event by applying function after synchronization
val wrap : event<'a> -> ('a -> 'b) -> event<'b>

// Creates a delayed event. When synchronizing on the returned
// event, the function returns an actual event to synchronize on
val guard : (unit -> event<'a >) -> event<'a>
```

We'll demonstrate working with `choose` and `wrap` in practice. The `guard` combinator allows us to create an event that performs some action only when it is synchronized on. The construct can be used for example for sending a request to a server. The following example (adapted from [61]) implements a thread that repeatedly reads a pair of numbers from two provided source channels (synchronizing on an input channel as soon as the number becomes available) and sends the addition of the numbers to the provided output channel:

```
1: fun add (inCh1, inCh2, outCh) =
2:   forever () (fn () => let
3:     val (a, b) = sync(choose [
4:       wrap (recEvt inCh1, fn a => (a, recv inCh2))
5:       wrap (recEvt inCh2, fn b => (recv inCh1, b)) ])
6:     in send(outCh, a + b) end
7:   )
```

The implementation uses a `forever` combinator that repeatedly runs the provided lambda function (line 2). Inside the body, we first need to obtain the two numbers that we want to add. This is done by synchronizing on an event composed using the `choose` combinator (line 3). The `choose` function makes a choice between two events. The event on line 4 synchronizes on receiving from the `inCh1` channel and then (using the post-synchronization combinator `wrap`) performs a blocking call to read a value from the `inCh2` channel. The second event (line 5) performs the same operations in the opposite order.

Manticore. When working with Concurrent ML, we need to write all operations using standard SML functions. This gives us a way for expressing useful abstractions, but it makes the syntax somewhat cumbersome. The Manticore language adds syntactic support for several common parallel programming patterns. These patterns include data-parallel constructs such as *parallel arrays* (a construct for creating arrays based on the set-builder notation that is evaluated in parallel) and *parallel tuples* (expressions that initialize individual elements of the tuple run in parallel), but also constructs for parallel value bindings based on futures.

We'll demonstrate two constructs that are both internally implemented using futures, which is an abstraction implemented using the low-level Concurrent ML machinery. The first example shows the `pval` construct which is a variable binding that starts evaluating in background and is automatically synchronized on when we access its value later in the code. The following function takes a binary tree as an argument and multiplies all values stored in the tree recursively:

```
1: fun treeProd (Leaf n) = n
2:   | treeProd (Node (treeL, treeR)) =
3:     let pval futureL = treeProd treeL
4:         pval futureR = treeProd treeR
5:     in (futureL * futureR) end
```

The function uses pattern matching to distinguish two cases. When the tree given as the argument is a leaf, we immediately return the value stored in the leaf (line 1). When the tree is a node with two sub-trees (line 2), we need to recursively multiply elements in both of the sub-trees and then multiply the results. This is done using the `pval` construct which starts performing the recursive call in background (lines 3 and 4). When we attempt to multiply the two values (line 5), the current threads needs to synchronize and wait until both of the computations running in background complete.

One possible improvement that we may want to implement would be to add a short-circuiting behavior. When one of the computations finish earlier producing 0 as the result, we know that the overall result will be also 0 and we could return immediately. In Concurrent ML, this could be done using the `choose` combinator and Manticore makes this possible using the `pcase` (*parallel case*) construct:

```
1: fun treeProd (Leaf n) = n
2:   | treeProd (Node (treeL, treeR)) =
3:     pcase treeProd(treeL) & treeProd(treeR)
4:       of 0 & ? => 0
5:        | ? & 0 => 0
6:        | l & r => l * r
```

The structure of the function is the same as in the previous version, but the body of the case that handles a node is different. The `pcase` construct takes several computations, separated by '&' as the argument (line 3) and it consists of several parallel patterns that match against the computations. The last pattern (line 6) consists of two standard ML patterns that assign the result to a symbol and it may be called whenever both of the computations complete. The first two patterns (lines 4 and 5) use so called wildcard pattern written as "?". It denotes that the pattern may match even if the value of the computation is not available yet. The other pattern is a constant value "0", so the first two cases may run if one of the computations produces zero, regardless of whether the other computation has completed or not.

Note that the `pcase` construct is in many ways similar to *joins* that support pattern matching on the carried values. The difference is that `pcase` is implemented in terms of more primitive operations, but it has been shown that *joins* can be also implemented using other concurrency primitives such as software transactional memory [35]. The Concurrent ML and Manticore research relates to the work presented in this thesis in the following ways:

- The syntax for pattern matching on features provided by the `pcase` construct allows us to express patterns such as waiting or one or two futures depending on value produced by one of the futures and it uses familiar pattern matching syntax. Our extension from Chapter V has similar expressivity. In Chapter VI, we use it for waiting on combinations of events in a way similar to Manticore.
- Our language extension isn't bound to a specific programming model and can be used outside of the reactive programming field. In fact, it is possible to use it for programming with futures, which gives us the same programming model as the one provided by Manticore (without focusing only on parallelism).
- Manticore is implemented in terms of combinators provided by Concurrent ML such as `choose`. Our language extension works in a similar way. In order to use it with some programming model, the developer needs to provide two primitive constructs (and one of them is similar to `choose` in CML)

6. Reactive languages

The languages discussed in the previous two sections focused on different types of programs than the ones we concern ourselves with in this thesis, but they shared some of the problems that we need to tackle. In this section we'll review languages and libraries that aim at the same domain as we, but use a different approach.

We'll start by looking at projects that originated from the influential work on Functional reactive animations (Fran) by Elliott and Hudak [1] who presented a purely functional way of describing animations in a declarative way. Then we'll look at an alternative approach which is also embedded in functional Haskell, but uses monads to implement an imperative programming model and finally, we'll look at a recent project by Meijer [5], which implements a variation on Fran using the LINQ project, which is now a part of the C# language.

6.1 Functional reactive programming

Programs written using any FRP library are written using two concepts. The first concept is called behavior and represents a value that is defined at any point and may change with the time. The second concept is called an event and represents an action that happens (possibly repeatedly) at some precisely specified time. In FRP solutions, the time is modelled as continuous.

The notion of continuous time allows some elegant constructions. For example, we can define a constant behavior that has a value 1 at all times. Then we can use a combinator for integration of numeric behaviors and the result will be a behavior that represents the current time. On the other hand, continuous time makes the implementation more difficult as we need to guarantee that the runtime will handle all events (which occur at a specific time).

FRP frameworks generally provide several ways for converting events to behaviors and vice-versa. We can for example declare a behavior whose value is the value carried by an event when it occurred the last time. In the other way round, we can define an event that is triggered whenever a value of a behavior changes or when it reaches some threshold.

Fran. We'll start by looking at the library called Fran [1], which is the basis of all functional reactive frameworks, but we'll use an F# implementation presented in [62, Chapter 15], which focuses on working with behaviors. A behavior is a value of type `Behavior<'a>` and represents a value of type `'a` that may change with the time.

For example, a value `Behavior<float32>` represents a floating point number that can change. This is a useful concept for creating animations, because if we create an ellipse whose location is specified as a value `Behavior<float32>`, we get an ellipse that moves depending on the time. An animation is represented as a value of type `Behavior<Drawing>`, which simply means that an animation is a drawing that changes over time. Let's look at some of the primitive behaviors as well as functions and operators for working with behaviors provided by the library:

```
// Primitive behavior returning the sine of the current time
val wiggle  : Behavior<float32>
// Creates a constant behavior that always has the same value
val forever : float32 -> Behavior<float32>

// At every time, multiplies the current values of the arguments
val (*)  : Behavior<float32> -> Behavior<float32> -> Behavior<float32>
// Composes two drawings by drawing the second one over the first one
val (--) : Behavior<Drawing> -> Behavior<Drawing> -> Behavior<Drawing>

// Result has a value that the input had before the specified time
val wait : float32 -> Behavior<float32> -> Behavior<float32>

// Moves the specified drawing by the specified offset. At every time,
// the current drawing will be moved by the current value of offsets
val translate : Behavior<float32> -> Behavior<float32> ->
                Behavior<Drawing> -> Behavior<Drawing>
```

The above primitives can be used for constructing many interesting animations. We can use overloaded multiplication operator to create a behavior that oscillates between larger values. For example, `wiggle * (forever 100.0f)` creates a value that oscillates between -100 and +100. The following code demonstrates how to create an animation that displays a sun in the center and the Earth rotating around the Sun at some specified distance. Initially, `sun` and `earth` are two constant drawings:

```
1: let rotate dist img =
2:     let pos = wiggle * (forever dist)
3:     translate pos (wait 0.5f pos) img
4:
5: let solarSystem =
6:   sun -- rotate 160.0f earth
```

The `rotate` function takes a float value and some drawings. It starts by declaring a sinusoidal value oscillating in the specified distance (line 2). To create a circular movement, we need to use a value of the cosine function as the second coordinate. One way to do this is to simply delay the sinusoidal value by the half of the amplitude (line 3). Once we have a function for rotating drawings, we can define the solar system, which consists of the Sun and a rotating Earth (line 6). The two objects are composed using the `--` operator, which draws one image over the other.

Yampa. Yampa [63] is a modern implementation of Functional reactive programming in the Haskell language. It uses arrows [47], which is an abstract representation of specific types of computations that is particularly useful for reactive computations. In addition, Yampa uses arrow syntax [48] which makes it easier to write programs based on the arrow abstraction.

When programming in Yampa, we're composing computations of type `SF a b`, which represents a time-varying function that takes an input of type `a` and produces a result of type `b`. A computation that takes some input and creates an event that fires at some time, carrying a value of type `b`, is represented as `SF a (Event b)`. For example, when programming robots, we may want to create a computation that takes the current state of the robot as the input and returns a time-varying speed of the robot. To create a speed that has a value 10 until the robot gets stuck (which is signalized by the event `rsStuck`) and then stops the robot by changing the speed to 0, we can use the following code:

```
speed :: SF SimbotInput Speed
speed = (constant 10 &&& rsStuck) `switch` \() -> constant 0
```

The `switch` construct takes two inputs (written on the left hand side and separated by '&&&') and a single parameter (provided on the right hand side), which is a function. The first input is `constant 10` and it specifies the initial time-varying value of the speed and the second input is an event `rsStuck`. Whenever the event fires, the `switch` construct runs the provided function to obtain a new time-varying value. In our example, the new value will be `constant 0`, so the robot stops running as soon as the `rsStuck` event fires.

Although values of type `SF a b` (called *stream functions*) closely resemble ordinary functions of type `a -> b`, they are different. Firstly, stream functions may depend on some other aspect such as the time and secondly, stream functions are constructed only using a limited set of primitives. In order to make construction of stream functions easier, Paterson has designed arrow syntax [48]. The following example uses this extension to create a more sophisticated stream function that controls the speed of the robot. The speed is controlled using the `incrVelsEvt` event that is triggered when the user pushes some button to increase the speed of the robot:

```
1: speed :: SF SimbotInput Speed
2: speed = proc inp -> do
3:   rec e <- incrVelEvs -< inp
4:       v <- drSwitch (constant 1) -< (inp, e `tag` constant (v+1))
5:       returnA -< v
```

The arrow syntax starts with the `proc` keyword and is followed by an identifier that represents the input of a stream function (in our case, a value of type `SimbotImput`). We're using a recursive definition, so the next line starts with the `rec` keyword. The body of the stream function consists of several bindings of the form `e <- f <- i`. This means that we're taking some input `i`, pass it through a stream function `f` and (eventually) obtain a result `e`. In our example, we first define a value that is present whenever the incrVelEvs event fires (line 3). Next, we use a `drSwitch` function which takes an initial value of the speed as an argument (`constant 1`). The input for the function (written on the right hand side of "-<") is the original `SimbotImput`

value and an event that carries a new stream function to be used instead of the previous one. We use the `tag` function to replace the value carried by the event e with a new value, which is a stream function that takes the current robot speed and increments it by one.

Explaining the system Yampa is beyond the scope of this brief introduction. However, we have shown how Functional reactive programs can be composed from events and behaviors. The relations with our work are following:

- The part of Functional reactive programming libraries that deals behaviors is in many ways complementary to the reactive programming library presented in this thesis, because our main concern is working with events. However, we believe that behaviors could be implemented as an abstraction over events.
- Functional reactive programming provides a powerful set of combinators for reacting to events such as `switch`. However, to our best knowledge, there is no direct way for encoding a state machine with transitions driven by events in FRP. This motivated us to design an imperative addition to a reactive programming library (Chapter VI) that makes this naturally possible.

6.2 Imperative streams

While Functional reactive programs are modelled using a continuous time, the Imperative streams [3] library uses discrete time steps and is in many ways more similar to synchronous languages. Another difference is that Imperative streams directly support imperative operations. In FRP, all actions (such as controlling a robot) need to be returned as a result of some stream function and processed by a top-level loop.

In Imperative streams, we structure programs using streams. A stream is constructed by an (imperative) computation that may use various looping constructs, may be defined recursively and may perform side-effects. Most importantly, the computation that defines a stream may also produce values that will be added to the stream at the current (discrete) time.

The following example shows an imperative stream that counts the number of times a button was pressed. The mouse button is itself represented as a stream that produces a value when a button is clicked. The programming model is synchronous, so there is a global clock and all reactions to a mouse event are processed within a single tick.

```
1: countClicks :: St Int
2: countClicks =
3:   let loop numClicks =
4:       until (next mouseButton)
5:             (lift numClicks)
6:             (do putStrLnI0 "Click"
7:                 loop (numClicks + 1))
8:   in loop 0
```

As we can see from the type signature, the `countClicks` stream will contain values of type `Int` (line 1). It is constructed using a recursive function `loop` that stores the current number of clicks in a parameter (line 3). Inside the body of the function, we use the `until` combinator.

The stream constructed by calling `until streamCtrl streamOne streamTwo` behaves as `streamOne` until a value is produced in the `streamCtrl` stream. After that it starts behaving as `streamTwo` and continues like this forever. In our example, the initial stream is `lift numClicks` (line 5) which is a stream that produces the value passed as the argument at the time when it is started and then does nothing. Once a value is produced in the `mouseButton` stream, the second stream provided to `until` is started. It prints the string "Click" (line 6) and then recursively invokes the loop function to produce the new count and start waiting for another click.

The notion of stream is somewhat similar to the notion of event that we use as the basic building block in this thesis. Our approach is to combine declarative and imperative approach for working with events and Imperative streams are closely related to the imperative part of our solution:

- The computation that constructs imperative stream forms an additive monad (meaning that it implements the `MonadPlus` type class). The `lift` operation corresponds to the `yield` primitive that we would use in F#. Our imperative computation from Chapter VI follows the same pattern, but it uses a different implementation of the `bind` primitive, which makes the programming model in many ways different.
- One similarity that is particularly interesting and that was demonstrated in the previous example is the use of recursive functions for creating streams. The pattern where we declare a recursive `loop` function that describes a stream and start it with an initial value as the argument will be used in many places in the Chapter VI.
- A notable difference between our solution and Imperative streams is that we don't use the synchronous programming model. In imperative streams, this aspect is very important as it allows an implementation of `bind` that takes the current value of a stream at each tick.

6.3 Reactive Extensions for .NET

The last library that builds on top of the concepts developed by Fran is Microsoft's Reactive Extensions for .NET [9]. Similarly to our work, the library focuses on working with events. It builds on top of the LINQ project [7], which extends the C# language with syntax for writing queries. The syntax has been used mainly for working with in-memory collections of data and with databases, but it isn't tied to a specific programming model and it is possible to define an implementation working with any data type.

Reactive Extensions for .NET provide an implementation that works on top of the `IObservable<'a>` type, which is similar to the `IEvent<'a>` type discussed earlier. The query syntax is used for constructing new, derived events from basic events provided by the system such as mouse events. We can use simple operations such as filtering and projection (which is similar to working with events using higher-order functions in F# as discussed in section 2.2). However, Reactive Extensions for .NET provide larger number of operators including several operators that can be used for combining values from multiple events in a way similar to joins.

The following example shows a query that implements drawing of rectangles. The user starts by pressing the mouse button and then moves with the mouse cursor to draw a rectangle. The drawing is stopped by releasing the mouse button:

```
1: var rectangles =
2:   from start in mouseDown
3:   from move in mouseMove.Until(mouseUp)
4:   let rc = CreateRectangle
5:      (start.EventArgs.X, start.EventArgs.Y,
6:       move.EventArgs.X, move.EventArgs.Y)
7:   select rc;
8:
9: rectangles.Subscribe(DrawRectangle);
```

The meaning of the `from` clause is that it handles all occurrences of the event that we take values from and runs the rest of the computation for each occurrence. In the example above, we first wait for the `mouseDown` event (line 2). For each of the occurrence, we start processing `mouseMove` events, but only until the `mouseUp` event occurs (line 3). This means that the subsequent lines will be executed for each `mouseMove` event that follows some `mouseDown` event, but before the mouse button was released.

The values carried by the events store information about the cursor location, so we use them to create a rectangle from the location of the `mouseDown` event to the current `mouseMove` event (lines 4 – 6). Then we trigger the constructed event with the rectangle as an argument using the select clause (line 7). Finally, we register a method that draws the rectangle as a handler for the constructed event.

- One difference between `IObservable<'a>` and `IEvent<'a>` is that the former type provides mechanism for notifying the user that it stopped producing events and will never occur again. This makes it possible to implement operations such as concatenation of two event streams. We use a relatively simple extension of the `IEvent<'a>` type that adds support for this notification.
- Reactive Extensions provide numerous operators for joining events. In a non-synchronous programming model, there are several possible semantics and Reactive Extensions provides extensive review [64].
- Similarly to Imperative streams and our imperative programming model, Reactive Extensions are also (in some way) based on monads, because the operation that is needed for supporting queries with multiple `from` clauses is similar to the monadic `bind`. Reactive Extensions use bind that calls the rest of the computation for each occurrence of an event, which is a different implementation than the one we use.

Chapter III

Approach and problem description

In the previous chapters, we introduced the area of reactive programming and we discussed numerous approaches for writing reactive programs in different types of environments. In this chapter, we present a high-level overview of the approach we develop in this thesis. We start by reviewing two reactive programming models. The first one is the declarative model (already discussed in section 2 of the previous chapter) and the second one is an imperative programming model. We'll discuss the benefits of the two models and we'll argue that it is beneficial to develop a single framework that makes it possible to combine the two approaches.

We'll briefly review some of the problems that we solve in this thesis. The two key problems are that, firstly, combining the two approaches to reactive programming in the naïve way makes it easy to introduce memory leaks and, secondly, when using the imperative programming model we need more expressive power to express waiting on different combinations of events. Finally, we discuss how our approach differs from the existing techniques presented in the previous chapter. Our approach is in many ways novel, so we briefly introduce the more specific contributions of the thesis.

1. Reactive programming

In this section, we review the two approaches to reactive programming and we'll discuss how they complement each other. We'll present the imperative style using a simple encoding based on F# asynchronous workflows that we introduced in [62, Chapter 16]. As we'll see later, using asynchronous workflows isn't sufficient for more complex scenarios, but it can be nicely used to introduce the problem. Finally, we also demonstrate that our approach isn't by any means limited to the F# language – we'll present a re-implementation of a part of the library in JavaScript. Now, let's start by looking at the unifying concept of an *event*.

1.1 Event as the unifying concept

Events in F# appear as an abstract type which allows the user to register and unregister handlers. A handler of type `Handler<'a>`, is a wrapped function (`unit -> 'a`), with a support for comparison via reference equality. An event is declared as an interface type. A somewhat simplified declaration that we'll use in this thesis looks as follows:

```
type IEvent<'a> =
  abstract AddHandler    : Handler<'a> -> unit
  abstract RemoveHandler : Handler<'a> -> unit
```

A simple event keeps a mutable list of registered handlers and invokes all handlers from the list when it is triggered. The interface only represents a value that can be used for listening to an event, so when creating a simple event, we can for example create a class that implements the interface and provides additional method to trigger the event.

1.2 Declarative event handling

In the declarative style, we write code using an algebra of events (a combinator library) that allows us to compose complex events from simple ones. The following example demonstrates how the declarative approach looks in F#. We take a primitive event `btn.MouseDown` (representing clicks on a button named `btn`) and constructs a composed event value called `rightClicks`:

```
1: let rightClicks = btn.MouseDown
2:   |> Event.filter (fun me ->
3:         me.Button = MouseButtons.Right)
4:   |> Event.map (fun _ -> "right click!")
```

We're using the pipelining operator `|>` (also known as the reverse application), which takes a value together with a function and passes the value as an argument to the function. This means that the `MouseDown` event will be given as an argument to `filter` function and the overall result is then passed to `map`.

The `MouseDown` event carries values of type `MouseEventArgs` and is triggered whenever the user clicks on the button. We use the `filter` primitive (line 2) to create an event that is triggered only when the value carried by the original event represents a right click. Next, we apply the `map` operation (line 4) and construct an event that always carries a string value "right click!". This approach is similar to Fran [2] and has the following properties:

- **Composability.** We can build events that capture complex logic from simpler events using easy to understand combinators. The complexity can be hidden in a library, as we can create functions that construct composed events.
- **Declarative.** The code written using combinators expresses *what* events to produce, not *how* to produce them. This makes the code more readable and easy to describe formally (as we demonstrated in Chapter II).
- **Limited expressivity.** On the other hand, the F# combinator library is limited in some ways and makes it difficult to encode several important patterns (e.g. arbitrary state machine)[5].

1.3 Imperative event handling

In the imperative style, we attach and detach handlers to events imperatively. To create a more complex event, we construct a new event and trigger it from a handler attached to other events. In F#, we can embed this behavior into asyn-

[5] In many cases, this is possible, but there is no obvious "natural" encoding of a state machine (such as using mutually recursive functions).

chronous workflows [4]. A workflow allows us to perform long-lasting operations without blocking the program. Technically, an asynchronous computation is represented as a function that starts the operation. When it is started, it gets a continuation as an argument. The operation should invoke the computation when the operation completes and a result is available.

To work with events, we can define a primitive asynchronous operation `AwaitEvent`, which takes an event value, waits for the first occurrence of the event and then runs the continuation. The implementation relies on imperatively registering a handler when the asynchronous operation starts and unregistering it when the event occurs for the first time.

The following code listing shows the implementation of `AwaitEvent`. The operation takes an event value as an argument, waits for its first occurrence and then resumes the workflow (at most once) giving it the value carried by the event as an argument:

```
1: let AwaitEvent (e:IEvent<'a>) : Async<'a> =
2:   Async.FromContinuations (fun (cont, _, _) ->
3:     let rec hndl = Handler(fun sender arg ->
4:       e.RemoveHandler(hndl)
5:       cont arg)
6:     e.AddHandler(hndl))
```

`AwaitEvent` constructs a primitive asynchronous operation using the `FromContinuation` method. The provided lambda function is called with a continuation `cont` as an argument when the workflow starts. Inside the lambda function, we create a handler (line 3); register it with the event and then return. When the event fires (at some later time), we remove the handler (line 4) and then invoke the continuation given as an argument (line 5) to run the rest of the workflow.

Now, we can use this primitive to get a powerful imperative programming model which is similar to Imperative Streams [3] or the Esterel language [11, 12]:

```
1: let clickCounter = new Event<int>()
2:
3: let rec loop count = async {
4:   let! _ = btn.MouseDown |> Async.AwaitEvent
5:   clickCounter.Trigger (count + 1)
6:   let! _ = Async.Sleep 1000
7:   return! Loop (count + 1) }
8: loop 0 |> Async.StartImmediate
```

The listing shows a function `loop`, which asynchronously waits for an occurrence of the `MouseDown` event (line 4) and then triggers the `clickCounter` event (created on line 1) with the incremented number of clicks as an argument. Next, it asynchronously waits one second and recursively calls itself and starts waiting for another click. Finally, we imperatively start the loop (line 8). Implementing the same functionality using event combinators is possible, but it leads to code that is much harder to understand. For curiosity, we included this implementation in the Appendix B.

When the `AwaitEvent` operation completes the handler registered with the `MouseDown` event is unregistered, so all clicks that occur while sleeping (line 6) are

ignored (there is no implicit caching of events). This means that the code shows a counter of clicks that limits the frequency of clicks to 1 click per second. In general, this approach has the following properties:

- **Imperative.** The code is written as a sequence of operations (e.g. waiting for an event occurrence) and modifies the state of events by registering and unregistering handlers or by triggering events.
- **Single-threaded.** Code in this style can be single-threaded using cooperative multi-tasking implemented using coroutines. This makes the concurrency in the model deterministic.
- **Composable.** Even though the implementation is imperative, the created event processors can be easily composed. In the listing, we constructed an event value `clickCounter`, which can be published, while the `loop` function remains hidden.
- **Expressive.** We can easily encode arbitrary finite state machines using mutually recursive functions (using the `return!` primitive). In the example, we have a simple case with just two states: 1) waiting for click and 2) waiting one second.

So far, we demonstrated the reactive programming model using the F# implementation. However, the approach is by no means limited to F# or some highly specific programming environment. It mainly relies on the support for higher order functions, which is now present in many languages including C# 3.0, Scala, Python, Ruby, but also for example JavaScript.

1.4 Compositional events in other environments

An implementation of event-based programming model is already available for C# 3.0 [5] and JavaScript library [24] builds on similar ideas. We created a simple implementation of F# event combinators in JavaScript (available on our web site [27]). The following example shows JavaScript version of the code from section 2.2. Note that this is very similar to code written using the, nowadays very popular, declarative jQuery library [25]:

```
1: var rightClicks = $("btn").mouseDown.
2:   filter (function (me) {
3:     return me.button == 2; }).
4:   map (function (_) {
5:     return "right click!"; });
```

The listing starts by accessing a primitive event value representing clicks on a button (line 1). The event value provides a `filter` function for filtering events and `map` for projection. We use them to create an event that carries the specified string value (line 5) and is triggered when the button value is 2, corresponding to the right click (line 3). As JavaScript doesn't have any equivalent to asynchronous workflows from F#, the imperative example would be slightly more complicated, but it can be implemented as well.

2. Issues and limitations

The previous section introduced an imperative programming model based on the `AwaitEvent` primitive, which works in a very limited scenario. We now discuss some of the issues and limitations of the presented approach.

2.1 Combining event handling techniques

We can view the declarative programming style of event processing as a higher-level approach. It allows us to write a limited set of operations in a way that is succinct, elegant and easy to reason about. On the other hand, the imperative style is lower-level, but as the previous discussion shows, it is extremely important for the ability to easily express state machines.

Now that we have two complementary approaches, it seems like a perfect solution to combine them and use the one that's more appropriate for the part of the problem that we need to solve. Unfortunately, combining the techniques brings some important implementation challenges.

When using the declarative style alone, we don't concern ourselves with removing handlers, because the event processing pipeline remains active during the entire application lifetime. As a result the present implementation of the higher-order functions for working with events never removes a handler from the source event. This means that the following example causes memory leaks:

```
1: let rec loop count = async {
2:   clickCounter.Trigger count
3:   let! me =
4:     btn.MouseDown
5:       |> Event.filter (fun me -> me.Button = MouseButtons.Right)
6:       |> Async.AwaitEvent
7:   return! Loop (count + 1) }
```

We implemented a simple imperative loop that uses `Event.filter` to wait only for click events caused by the right mouse button (line 5). The `Event.filter` function adds a handler to the `btn.MouseDown` event during every iteration of the loop, but the handler is never removed, even though the `AwaitEvent` primitive correctly unregisters a handler from the source event (in our example, the event returned by the `Event.filter` function).

2.2 Limited expressivity of AwaitEvent

The `AwaitEvent` primitive makes it possible to wait for the first occurrence of a single event inside an asynchronous workflow. However, in many situations we need to wait for one of several events, for a subsequent occurrence of two events or for an occurrence that carries a specific value as an argument.

In [62, Chapter 16], we added an overloaded version of the `AwaitEvent` that supports waiting the first of multiple events, which is perhaps the most frequently needed combination in practice. However, this solution is still very limited. We demonstrate it using a counter that supports resetting. It counts clicks on the `btn` button and can be reset by clicking on the `btnReset` button:

```
1: let rec loop count = async {
2:   clickCounter.Trigger count
3:   let! e = Async.AwaitEvent(btn.Click, btnReset.Click)
4:   match e with
5:   | Choice1Of2(m) -> return! loop (count + 1)
6:   | Choice2Of2(m) -> return! loop 0 }
```

The sample is again implemented using a recursive loop that stores the current count in the value of a parameter. When it runs it first reports the current count to the `clickCounter` event (line 2). Then it starts waiting for two events simultaneously (line 3). The overloaded version of `AwaitEvent` returns a value of type `Choice<'a, 'b>` to the asynchronous workflow when the first of the two events occurs. The result is tagged with a label `Choice1Of2` when the first event was triggered first or with the label `Choice2Of2` in the opposite case. Once we obtain the value, we use pattern matching to choose the appropriate reaction. If the user clicked on the counting button, we increment the count (line 5) and if the event was caused by the reset button, we set the count to zero (line 6).

The overloaded version of `AwaitEvent` handles only one specific scenario and if we want to use it, we need to explicitly pattern match on somewhat mysterious labels `Choice1Of2` and `Choice2Of2` that lack any clear meaning. We would like to avoid this and use convenient syntax, possibly similar to joins or the `pcase` construct from Manticore.

3. Approach description

In this section, we give a brief high-level overview of the approach we use to solve the discussed problems. In addition to combining of the declarative and imperative programming models and limited expressivity of the `AwaitEvent` primitive, we also introduce the monadic computation we use instead of asynchronous workflows.

3.1 Mixing declarative and imperative

As we'll see, there is nothing fundamentally wrong with the declarative programming approach that would make it impossible to create an implementation that would work when combined with the imperative approach. However, we need to carefully analyze the situations that may occur when combining the two approaches.

One possible solution would be to design a specialized garbage collection algorithm for the reactive programming scenario. This approach would be probably the safest possible as it would transfer the responsibility from the user to the runtime. This would be just one more step in the direction that is followed by most of the modern high-level programming systems (where garbage collection automatically reclaims unused passive objects). However, implementing a GC algorithm isn't a solution viable from the practical point of view.

Instead, we'll present a mental framework that makes reasoning about the problem easier and allows us to design a correct implementation of standard F# combinators. Since users don't (usually) need to provide their own combinators this solves the problem once and for all without placing any requirements on the side of the user.

3.2 Pattern matching on events

One way to encode more complicated patterns of waiting for events is to use a combinator library. This technique is used by Concurrent ML when expressing synchronization on events. The benefit of this approach is that it doesn't require

any language extension, because combinators are just functions or custom operators. On the other hand, we find this solution somewhat cumbersome to use.

Our solution is to design a syntax based on pattern matching, which is a familiar construct in functional languages. The user can match on multiple events and can provide multiple clauses. A clause can be triggered only when the required events produce some value and when the value matches the required pattern. This way, we don't get a fully general language for specifying combinations of events, but we get a simple and easy to use solution that works very well in most of the practical situations.

One problem with languages based on Join calculus or with Manticore is that they are very specific and support only a single programming model. We believe that this is the reason why these techniques aren't (so far) used by mainstream languages. To avoid the danger of being too specific, we design an extension that is general purpose and can be used not only in the reactive programming scenario, but also for various concurrent programming models. In particular, we'll show that it can be used for encoding Manticore-style working with futures and concurrency model based on the Join calculus.

3.3 Event builder computations

When using the imperative programming model based on asynchronous workflows, we need to explicitly construct an event and then trigger it imperatively from a workflow (which itself has to be imperatively started). To provide a more straightforward way for constructing events imperatively, we design a language based on the F# computation expression syntax.

We call the construct *event builder computation* and the result of using it is a value of type `IEvent<'a>` and a running process that can trigger it. Eevent builder computations have the following properties:

- **Yielding values.** Event builder computations provide the `yield` construct for triggering the constructed event. This is similar to the `lift` primitive of Imperative streams and similarly to imperative streams, our `yield` constructs represents the return operation of a monad.
- **Imperative monad.** Our computation is based on the computation expression syntax but, strictly speaking, doesn't conform to any of the common mathematical structures that are usually used with computation expressions. The motivation for this is that our bind operation implements imperative waiting for the first occurrence of an event, which breaks some of the usual monad laws. However, we believe that our type of computation is interesting on its own. We'll establish a set of laws that hold about it and present another useful example of this type of computation.
- **Semi-discrete time.** Our programming model is neither synchronous (as in synchronous languages or Imperative stream), nor based on the continuous time (as in FRP). We develop a notion of semi-discrete time that makes it possible to express the fact that a reaction to an event may trigger multiple events in response, but that makes the model deterministic without relying on sophisticated implementation techniques (such as interval analysis in FRP).

To briefly summarize the most important points of our approach, we'll conclude the chapter by looking at a basic example that demonstrates several of the important aspects discussed in this section.

4. Sample solution

To demonstrate the approach developed in this thesis, we'll look at one part of an application for drawing rectangles. This is an interesting problem because the application needs to react to several combinations of events. When discussing Reactive Extensions for .NET, we implemented a part that is responsible for an immediate feedback when drawing. Here, we'll look at a different component.

We'll implement an event that triggers once after the user finishes drawing a rectangle. In response to this event, we could for example store the rectangle in a list of all drawn shapes. Note that this is a part of a larger example that is discussed in Chapter VI, so other components of the application will be discussed later. To make the example more interesting, we also handle cancellation of the drawing. When the user starts drawing (by pressing the mouse button) and then hits the Esc key before releasing the button, the rectangle will be ignored:

```
1: let rectangles =
2:   let rec loop() = event {
3:     let! down = form.MouseDown
4:     match! form.MouseUp, form.KeyDown with
5:     | !up, !ke when ke.KeyCode = Keys.Escape ->
6:         yield! loop()
7:     | !up, _ ->
8:         yield Rectangle(down.X, down.Y, up.X, up.Y)
9:         yield! loop() }
A:   loop()
```

The expression declares an event of type `Event<Rectangle>` named `rectangles`. The event will be triggered when the user successfully enters a rectangle. To construct the event, we use event builder computation that uses a recursive function named `loop` (line 2). The computation first starts by waiting for the first occurrence of the `MouseDown` event (line 3). When the event occurs, we need to wait for the `MouseUp` event, but we also need to handle the case when Esc key was pressed in the meantime. This can be implemented using our extension that allows us to write pattern matching on events.

We match on the `MouseUp` and the `KeyDown` events (line 4). The first clause (line 5) can run only when the `KeyDown` event fires carrying a value representing the Esc key and when the `MouseUp` event fires (in an arbitrary order). The second clause (line 7) requires only the occurrence of `MouseUp` event. This means that when the MouseUp event fires, we can always select one of the clauses to continue (and cancel the other). If the Esc key was pressed in the meantime, we select the first clause, because our extension preserves the semantics of ML pattern matching where the order of clauses is important.

When the Esc key was pressed, we recursively continue looping without generating a rectangle (line 6). If the user draws a rectangle without clicking Esc, then

we use the yield construct to trigger the constructed event (line 8) with the newly created rectangle as the argument and then continue looping to wait for another `MouseDown` event. The example shows the following aspects of our approach:

- We demonstrated how the event builder computations look. The notable aspects are that we can use `let!` construct for waiting for a single occurrence of an event and that we can use the `yield` construct to trigger the constructed event. The fact that we're waiting only for a single occurrence of an event (or a first viable combination of events) means that the typical pattern used when writing a computation is to use recursive functions (that are suitable for encoding state machines).
- Our approach is imperative, which means that the computation needs to reach the `let!` construct first, in order to handle the next occurrence of an event. However, we guarantee that (under some well-defined conditions) the processing of the reaction will complete before the next event can occur. This is the key idea behind our concept of semi-discrete time.
- Finally, the example also demonstrated generalized pattern matching on events. We've seen that it can be used for specifying which events must occur before a clause can run and the example also shows that we can specify some condition on the value carried by the event. We believe that the above example shows that pattern matching syntax is more convenient than a solution based on combinator libraries.

Chapter IV

Garbage collection for reactive programs

In this chapter, we focus on the problem of garbage collection in the reactive programming scenario. For events, we need to consider not only whether the event value is reachable, but also whether it can have any effect. An incorrect treatment can lead to unexpected behavior and can cause memory leaks when combining the two styles of reactive programming. In particular, the key contributions of this chapter are the following:

- We state which events are garbage and show that this definition is dual to the notion of garbage for objects. We compose these two concepts into a single one that is useable for an environment containing both events and objects (Section 2). To build a better intuition, we present a formal algorithm for garbage collection (Section 3) in this environment.
- We present an implementation of combinators for declarative event-driven programming in F#, which does not suffer from memory leaks (Section 4) and we show how it follows from our formal model (Section 5).

Our approach is pragmatic, so we target mainly established platforms. We aim to show how to develop a correct, memory-leak free reactive library without modifying the garbage collection algorithm and using only features available at most of the platforms. The GC algorithm presented in this chapter serves mainly as a useful mental model for reasoning about the problem.

1. Problems with mixing styles

We already briefly introduced the problem that appears when we try to mix the declarative and imperative styles in Chapter III. We discuss it in a more detail in this section, but first, we briefly introduce some aspects of our programming model that are important for understanding how to treat events with respect to garbage collection.

1.1 Event-based programming model

In our reactive library, events are values like any other. In the formal model, we'll distinguish between these two constructs. This allows us to treat *objects* and *events* differently in the garbage collection algorithm. This is desirable as events are in many ways special.

Private references. One notable property of events is that all references to other events or objects are private. They are captured in a closure and cannot be accessed by the user of the event. This has an important practical implication for our implementation. If an event e_1 references event e_2 and object o_1 references e_1, we cannot directly access the event e_2 from code that uses o_1.

Created by combinators. When developing the reactive programming library, we'll use the described garbage collection techniques only for collecting events that are created by declarative event combinators. Notably, due to the limited set of combinators, this guarantees that there won't be any cyclic references between events. Our formal model (Section 4) is fully general, but our reactive library (Section 4) takes advantage of this simplification. In the absence of cycles, we can safely use a variant of reference-counting in the library implementation.

Side-effects. Our programming model can be embedded in an impure functional language, so the predicates provided as parameters to combinators (e.g. `Event.map`) may contain side-effects. It is not intuitive when and how often the side-effects should happen, so they are discouraged. However, we make the following very weak guarantees that are fulfilled by both implementations discussed in this chapter as well as [5]:

- When a handler is attached to an event created by a combinator with an effectful predicate, the side-effect is executed *one or more times* when the source event occurs.
- When no handler is attached to the event, the effect may be executed *zero or more times* when the source event occurs.

Next, we'll explore an example that motivated the work presented in this chapter. It will clarify which events should be garbage collected.

1.2 Disposing processing chains

It is possible to implement event combinators using a very simple pattern[6]. In this pattern, each combinator creates a new event (a stateful object that stores a list of event handlers) and registers a handler to the source event that triggers the created event:

```
1: let map f (src : IEvent<_>) =
2:   let ev = new Event<_>()
3:   src.AddHandler(Handler(fun x ->
4:     ev.Trigger(f x)))
5:   ev.Publish
```

The listing shows the `Event.map` combinator. It registers a handler (lines 3, 4) to the source event and when the event occurs, it applies the function `f` to the carried value and triggers the created event. As we can see, it never unregisters the handler that was attached to the source event using the `AddHandler` member.

Let's demonstrate what exactly happens when we create an event processing chain using several combinators, add an event handler, wait for the first occur-

[6] This pattern has been used in F# libraries including version 1.9.7.8, which is the latest version available at the time of writing this thesis.

rence of the event and then remove the handler. This behavior is just a special case of what we can write using the `AwaitEvent` function:

```
1: let awaitFirstLeftClick src k =
2:   let clicks = src.MouseDown
3:     |> Event.filter (fun m ->
4:          m.Button = MouseButtons.Left)
5:     |> Event.map (fun m -> (m.Y, m.X))
6:   let rec hndl = new Handler<_>(fun arg ->
7:     clicks.RemoveHandler(hndl)
8:     k arg)
9:   clicks.AddHandler(hndl)
```

The function takes a continuation `k` and a source event `src` as parameters. It uses event combinators to create an event that is triggered only when the source event was caused by a left click (lines 2 to 5). On the line 6, we create a handler object (using F# value recursion [20]). When it is called, it unregisters itself from the event and invokes the continuation (lines 7, 8). Finally, the function registers the handler returning a unit value as the result.

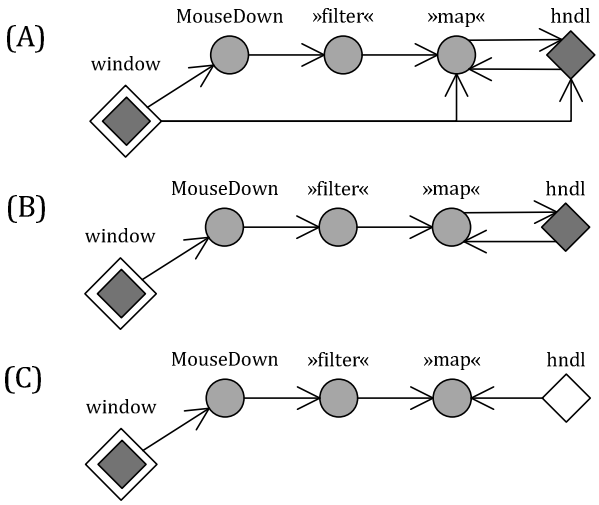


Figure 6. References in the processing chain after it is created (A); when references from `window` are lost (B); after the handler is removed (C); events are shown as circles, objects as diamonds; window is marked as a root object; white color means that object or event is not referenced (and will be garbage collected)

The Figure 6 shows objects and references between them, which are created by running the previous function. When constructing the event `clicks`, each combinator creates a reference from the source event to the newly created event (by registering an event handler). The initial situation, right after running the line 7, is shown in (A). The executing program still keeps references to the local values of the function (`hndl` and `»map«`) and the closure of the `hndl` value references the event `»map«` (captured because of the reference on line 5).

We can see what happens when the function returns in (B). The stack frame for the call is dropped, so the root object loses direct references to the constructed event and the handler. At this point, we don't want to dispose any part of the chain! It can still do something useful (run the handler and trigger the continuation). As

the diagram shows, there are still references from `window` to all events, so the implementation above behaves as we need.

The situation displayed in (C) is more interesting. When the event occurs, the handler unregisters itself from »`map`«. There are no other references to `hndl` and it can be garbage collected. The rest of the processing chain isn't disposed, because it is still referenced from the root. Arguably, the chain cannot do anything useful now, as there are no attached handlers. When the source event occurs, it will only run the specified predicates, which is allowed, but not necessary (as discussed in 3.1). More importantly, when we add and remove handlers in a loop (e.g. the example in section 2.2), we'll create a large number of abandoned events that cannot be garbage collected. Obviously, this isn't the right implementation.

2. Garbage in the dual world

Due to the inversion of control, event-driven applications are in a way dual to "control-driven" applications. To our knowledge, this duality hasn't been described formally in the academic literature, but it has been observed by Meijer [5]. He explains that a type representing events is dual to a type representing sequences. Interestingly, we can use the principle of duality when talking about garbage collection in the reactive scenario as well.

2.1 Garbage in worlds of objects and events

In section 1.2, we've discussed a case where an event intuitively appeared to be useless, but wasn't disposed by the GC, because it was still referenced. This example suggests that we need a different definition of "garbage" for reactive applications. Formally, we model references between objects as an oriented graph $G = (V, E)$ consisting of *vertices* V and *edges* E. A set of *roots* $R \subseteq V$ models objects of a program that are not the subject of garbage collection (such as objects currently on the stack etc).

A vertex $v \in V$ is *object-reachable* if and only if there exist a path $(v_o, \dots, v_n, v)$ where $v_o \in R$. Objects that are garbage are those that are not *object-reachable*. (1)

Let's now focus on events. In section 1.2, we stated that events are useless if they cannot trigger any handler. We'll define this notion more formally. We take *leaves* $L \subseteq V$ to be events with attached handlers. Then the event value is useful if we can follow references from it and reach one of the leaf events (meaning that triggering of an event can cause some action). If we were in a world where everything is an event and the events are triggered from the outside, then we could use the following definition:

A vertex $v \in V$ is *event–reachable* if and only if there exists a path $(v, v_o, \dots, v_n)$ where $v_n \in L$. Events that are garbage are those that are not *event-reachable*. (2)

We can observe that the definition of *event-reachable* (2) is equivalent to the definition of *object-reachable* (1) in the inverted reference graph (taking *leaves L* as

roots R). This explains why we were referring to the duality principle in the introduction of this section. The reactive world isn't dual only when it comes to types (as noted by Meijer), but also when it comes to the definition of garbage.

2.2 Garbage in the mixed world

In practice, we're working with an environment that contains both objects and events. When collecting garbage, we want to mix the two approaches outlined in the previous section. We want to follow the *object-reachable* definition for objects and *event-reachable* definition for events.

Combining the two notions requires some care. We will take the *roots R* of the graph to be the root *objects*, but how can we incorporate events? In this section, we'll look at an intuitively clear way to define collectability for a mixed world. We start by distinguishing between objects and events:

> Let vertices V be a union of two disjunct sets $V_e \cup V_o$ where V_e is the set of *events* and V_o is the set of *objects*.

Events in the mixed environment aren't triggered from the outside, but by other *events* or *objects* that reference them. This means that events that are not *object-reachable* are also garbage in the mixed world. A more subtle problem is determining *leaf events* that "are useful". We will explain the definition shortly:

> We define the set of leaf events T as follows:
>
> $$T = \left\{ v \in V_e \;\middle|\; \begin{array}{l} \exists \mathrm{v}_0 \in V_o : v_o \text{ is } object\text{-}reachable \\ \text{and } \big((v_o, v) \in E \vee (v, v_o) \in E\big) \end{array} \right\} \quad (3)$$
>
> An object or event $v \in V$ is *collectable* if and only if it is not *object-reachable* given roots R or if it is an event ($v \in V_e$) and it is not *event-reachable* given leaves T. (4)

As already discussed, we check *object-reachability* of both events and objects (4). For events we apply an additional rule, using a constructed set of event *leaves T*. The elements of T in (3) are defined as a disjunction of two conditions. The first one specifies *events* that are directly referenced by some *object*. In this case, we mark them as "useful", because they can be directly accessed by program and the program may intend to register a handler with them at some later point. The second condition specifies *events* that directly reference some object, which corresponds to the fact that there is some registered event handler.

The definition is demonstrated by Figure 7. As we can see, all events directly referenced by objects or referencing an object (handler) are marked as leaves. Using the first part of (4), we mark objects and events in the lower part as garbage, because they are not referenced from the root object. The upper part demonstrates events that become garbage due to the second part of the definition. They are referenced from the *root object*, but there is no path leading to any *leaf event*.

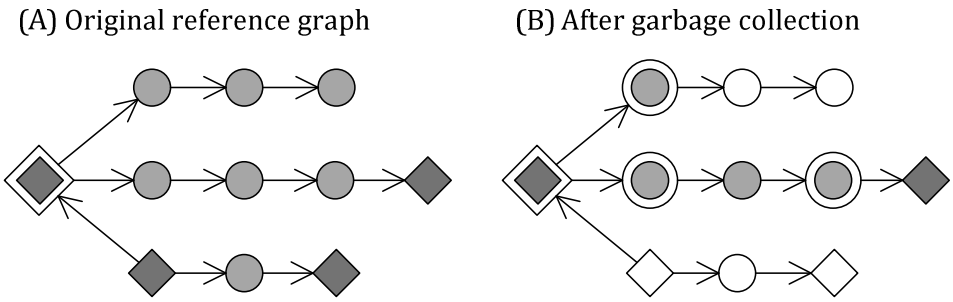


Figure 7. Mixed garbage definition. (A) shows initial reference graph that mixes objects (diamonds) with events (circles); root objects and calculated leaf events are marked with rings. (B) shows which objects and events are garbage (filled with white color).

The next section describes a garbage collection algorithm for the mixed environment of objects and events, taking advantage of the aforementioned duality.

3. Garbage collection algorithm

Implementing a specialized GC algorithm is a difficult task in practice, so we instead describe how to build an algorithm using a standard GC algorithm for collecting objects that are not *object-reachable*. Such algorithms are already well-understood and are implemented and optimized on many platforms.

3.1 Constructing the algorithm

The input of our algorithm is a reference graph $G = (V, E)$ with a set of root objects R. It works in three steps. The first two steps perform pre-processing dealing with the integration of environments and the third step uses the duality principle.

Pre-collection. As already discussed, only events that are referenced by an object or another event can be triggered. As a result, we first need to collect objects and more importantly also events that are not *object-reachable* using (1). This corresponds to running a GC algorithm on the original reference graph containing both events and objects. This can be described as follows:

$$V^{pre} = \{v \in V | v \text{ is } object\text{-}reachable\}$$
$$(V^{pre}, E^{pre}) = G^{pre} = G[V^{pre}] \quad (5)$$

The graph G^{pre} is a subgraph of G induced by the reduced set of vertices V^{pre}. We'll also need a reduced set of object vertices $V_o^{pre} = V^{pre} \cap V_o$ and event vertices $V_e^{pre} = V^{pre} \cap V_e$. We can easily see that the constructed graph doesn't contain any *collectable* objects or events as defined by the first part of (4).

Mock references. From this point, we want to treat events separately from objects, so events will no longer keep other objects alive by referencing them. To make sure that all *object-reachable* will remain *object-reachable* we add *mock references* to simulate chains of events. We'll add references from event sources (objects that reference events) to all event handlers (objects that are referenced by events) that are reachable by following only event vertices. More formally:

$$E^{pre'} = E^{pre} \cup \{(v,u) | v,u \in V_o^{pre} : p_e(v,u)\} \quad (6)$$
$$p_e(v,u) = \exists path(v, v_1, \dots, v_n, u) : n > 0 \wedge \forall i : v_i \in V_e^{pre}$$

The predicate p_e states that there is a path between two vertices, which visits only *events* and its length is at least two. By adding edges to the graph (6), we ensure that all event handlers that can be triggered by an event will be referenced. Note that these event handlers correspond to the second part of the condition specifying leaf events T in the definition (3).

Duality principle. The previous two steps performed pre-processing that is necessary when we want to integrate events and objects. Now we can use the key idea of this article, which is the duality between the definitions of *object-reachable* and *event-reachable*. We construct a transformed *garbage graph* G^*:

$$G^* = \left(V^{pre}, \{d(e) | \, e \in E^{pre'}\}\right) \text{ where} \quad (7)$$
$$d(u,v) = \begin{cases} (u,v) \; when \; u \in V_o^{pre} \\ (v,u) \; when \; u \in V_e^{pre} \end{cases}$$

The function d reverses references leading from an event to any other vertex. It unifies the notion of *object-reachable* from (1) and *event-reachable* from (2). This allows us to handle the second part of the collectability definition (4) using a standard GC algorithm for passive objects. Finally, we run it on the *garbage graph* G^*:

$$V^{fin} = \{v \in V^{pre} | v \text{ is } object\text{-}reachable \text{ from } R \text{ in } G^*\} \quad (8)$$

The definition (8) gives us the final set of objects and events that are not garbage. To get the final reference graph, we take the result of pre-processing (5) and take a subgraph induced by V^{fin}.

3.2 Garbage collection example

Before we discuss the correctness of the algorithm, let's look at **Figure 8**, which demonstrates the construction steps using a minimal example with most of the important situations.

The diagram (A) shows an initial state with a *root* object referencing an event source. The source can trigger a `MouseDown` event, which is propagated through the chain to a `handler`. During the pre-processing, we run garbage collection to remove events that cannot be triggered, which removes the `evt1` event and we also add a mock reference from `source` to `handler`, which allows us to reverse references of the chain (B).

Finally, (C) shows what happens after reversing the edges that lead from events. Thanks to this operation, we start applying the dual rule to events, so all events that cannot trigger any *leaf* event become garbage (e.g. `evt2` in the diagram). However, thanks to pre-processing, we don't affect collectability of non-events. The algorithm intuitively follows the definition, so it shouldn't be difficult to believe that it is correct. However, the following section presents a more formal proof of the algorithm correctness.

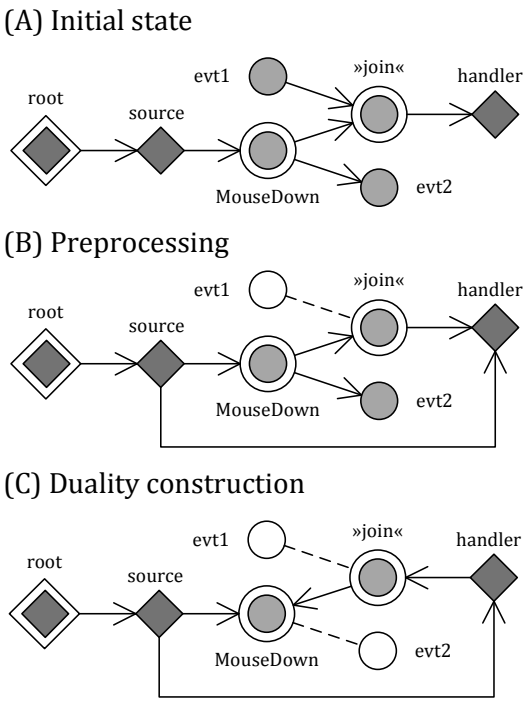


Figure 8. Graph construction. Objects are shown as diamonds and events are displayed as circles; dashed lines are references to events and objects that were garbage collected.

3.3 Correctness of the algorithm

To show that the algorithm is correct, we analyze two cases. First, if an object or an event is *collectable*, it will be collected by our algorithm and second, if an object or an event is not *collectable*, our algorithm won't collect it.

Case I.: Collectable. The definition (4) describes two cases in which an object or an event $v \in V$ is *collectable*. First, it may not be *object-reachable*, which means that there is no path from the roots R to it. In this case, our algorithm will collect v in the first pre-processing step (5), which simply collects all vertices that are not *object-reachable*. The second case is more interesting. An event $v_e \in V_e$ is *collectable* if it is not *event-reachable*, meaning that there is no path from it to any *leaf event* from T. This means:

There isn't any path[7] from v_e to an event that is referenced by an object (first condition in (3)) or to an object (second condition in (3)). (9)

Now, we need to show that v_e will not be *object-reachable* after we reverse the reference graph in (7).

- Firstly, all existing edges leading to v_e are from events (otherwise v_e would be in T), so they will be reversed and can't be a part of a path leading from any *root object* to v_e.

[7] This may also include paths of length 1 (with only a single vertex).

- Secondly, all edges leading from v_e will be reversed, which creates new paths leading to v_e. However, due to (9), all new paths will be only from events, and so they also cannot lead from any *root object*.

In summary, if the object or event was *collectable*, it will be collected in pre-processing. If an event wasn't *event-reachable* (but was *object-reachable*) it will not be *object-reachable* after we reverse the references and will be collected in (8).

Case II. Not collectable. On the other hand, let's take any object or event $v \in V$ that is not *collectable*. First of all, there must be some path to v from some *root object*, which means that it is *object-reachable*. As a result, it won't be garbage collected in the first pre-processing step (5), where we ensure that all objects and events are reachable. For the rest of the algorithm, we need to distinguish between two cases: when v is an *object* and when v is an *event*. In the first case, $v \in V_o$ and there is a path to it from some root object. We want to show that after reversing references in (7) there will still be a path to object v.

> We take the original path and replace any sub-path $(u, v_1, \dots\ v_n, w)$ such that $\forall i: v_i \in V_e$ and $u, w \in V_o$ with the mock edge (u, w). The mock edge exists, because the path matches the predicate $p_e(u, w)$ from (6).

The newly constructed path consists only of edges between objects, and so it will not be affected by any alteration of edges between events. As a result the object v won't be collected in (8). In the second case we have $v \in V_e$ that is not *collectable*, so there is a path from v to some *leaf* event $r \in T$ as defined in (4). After reversing the references, there will be a path from r to v. Now, we need to show that r won't be *object-collectable*.

- If the *leaf* event $l \in T$ was originally referenced by some *object-reachable* object $v \in V_o$, it will be still be referenced after reversing references (the edge (v, l) won't be modified). Moreover, v will still be *object-reachable* as shown in the first sub-case of this section.
- If the leaf event $l \in T$ originally referenced some object-reachable object $v \in V_o$, it will be referenced by this object after reversing references (because the edge (v, l) will be reversed). Just like in the previous point, v will still be *object-reachable*.

3.4 Removing events during collection

As noted earlier, we use the formal algorithm mainly as a formal model and it motivates the implementation of a reactive library in the next section, but first we briefly discuss an aspect that would be important for an actual implementation of the algorithm.

When removing an event from the memory (for example `evt2` in **Figure 8**), there won't be any references to it from objects, because that would make the event a leaf event and it wouldn't be collected. However, it may be referenced from other events. To avoid dangling references, we need to deal with these possible references when collecting the event.

Ideally, the garbage collector would have some knowledge about events and it could remove the reference to the collected event from the source event (for

example if an event in the actual system contained a list of referenced events). Another approach would be to redirect the reference to a special *null event*, which doesn't perform any action when triggered.

4. Implementing the reactive library

The algorithm proposed in section 3 has the advantage that it can be built on top of a standard GC algorithm, but we wanted to avoid modifying the runtime altogether. In this section, we discuss an implementation of combinators which is inspired by the previous discussion and implements almost identical behavior.

As discussed in section 1.2, a naïve implementation of event combinators registers a handler to the source event (using the `AddHandler` function). It keeps a mutable list of handlers and returns a new `IEvent<'a>` value, which adds or removes handlers to or from this internal list. When the source event fires the combinator chooses whether to trigger the handlers and also calculates a value to use as an argument. This implementation is faulty, because it never removes the handler from the source event.

4.1 Implementation requirements

Let's briefly summarize the requirements for the new implementation. Section 1.2 already discussed the most important aspect:

- **Collectable event chains.** When the user unregisters all previously registered handlers and when there is no other reference to the event value, the entire processing chain should be available for garbage collection. This corresponds to the definition of garbage for events from section 2.
- **No explicit referencing.** When there is a handler attached to the event value representing the chain, the chain shouldn't be collected as the handler can still perform some useful work.
- **Stateful.** As we'll clarify in section 3.2, the current semantics of event combinators is stateful, meaning that the mutable state of a combinator can be observed by multiple handlers. We want to preserve this semantic property.
- **Composability.** When we create a chain using a sequence of pipelined `Event.xyz` transformations, we should be able to freely split it into several independent pieces (e.g. we should not require adding a special combinator to the end or the beginning of the event processing pipeline).

The original implementation didn't meet the first requirement. However, when designing a library that satisfies the first condition, it is easy to accidentally break another one. The requirement for stateful combinators deserves more explanation.

4.2 Stateful and stateless model

Certain combinators maintain a state. The same state is shared by all handlers attached to a single event value. We can demonstrate this behavior using one more example that counts clicks:

```
1: let counter = btn.MouseDown
2:   |> Event.map (fun _ -> 1) |> Event.scan (+) 0
3:
4: let rec loop() = async {
5:   let! num = counter |> Async.AwaitEvent
```

```
6:   printf "count: %d" num
7:   return! loop() }
```

We take the `MouseDown` event and project it into an event that carries the value 1 each time the button is clicked. The stateful combinator `Event.scan` uses the second argument as an initial state (line 2). Whenever the source event occurs, it uses the function provided as the first argument (in our case +) to calculate a new state from the previous one and the value carried by the event. The returned event is triggered (carrying the current state) each time the state is recalculated.

Next, we switch to the imperative event handling style and implement a processing loop that prints the current count whenever `counter` event fires. The loop repeatedly attaches a handler to the event (line 5), so the code works only if the state stored by the event is shared between all event handlers.

Let's compare the two possible implementations of event combinators that maintain a state between two occurrences of the event (such as `Event.scan`):

- **Stateless.** In this model, we create a unique instance of the state for each attached event handler. This model keeps the code referentially transparent, but it works well only in a purely declarative scenario. If we ran the example above using a stateless implementation, it would print 1 for every click, which is somewhat unexpected.
- **Stateful.** The state of an event created by the combinator is shared between all handlers. This approach is consistent with the imperative event handling. And finally, as the previous example demonstrates, the two styles work very well together in this setting.

The stateless approach has its benefits, especially for pure languages. It has been used in Haskell [2] and is also being used by the Reactive Framework [5], which builds on LINQ [7]. We'll follow the pragmatic ML tradition and use the stateful implementation. The next section shows an implementation inspired by the formal algorithm that follows all the technical requirements.

4.3 Constructing event chains

We'll discuss the implementation by looking at an example similar to the problematic case from section 1.2, but we'll also analyze other important situations. We start by constructing an event chain and a handler that we'll later attach to the composed event:

```
1: let clicks = src.MouseDown
2:   |> Event.filter (fun m ->
3:        m.Button = MouseButtons.Left)
4:   |> Event.map (fun m -> (m.Y, m.X))
5:
6: let hndl = new Handler<_>(...)
```

Figure 9 (A) shows what happens after running this code. Our implementation doesn't attach any event handler to the original event source. Each event only keeps a reference to the source, so that it can attach the handler later. We call this *lazy initialization*. If we compare this diagram with the references in the naïve implementation (**Figure 6**, A), we can see that edges leading from events are reversed, which corresponds to our construction in (9).

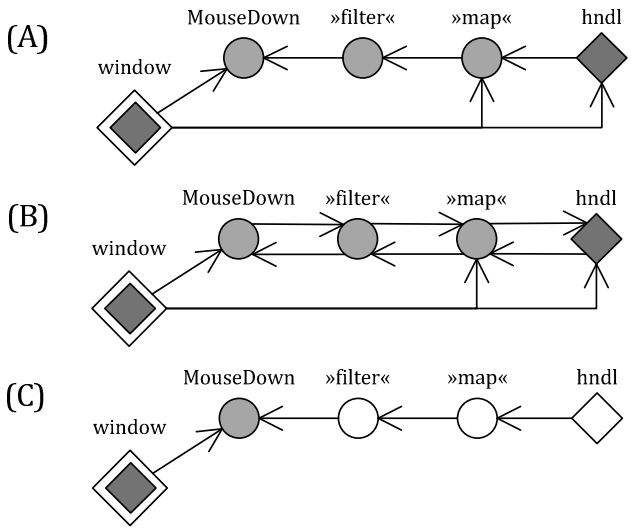


Figure 9. Implementation using reversed links. The state right after creating an event chain (A); after registering a handler, forward references are created (B). Removing the handler also removes forward references. If we also lose direct references from the program, the entire chain becomes collectable (C).

Adding event handlers. Now that the chain is initialized, let's look at what happens when we register `hndl` as a handler for the event constructed by the `map` function (displayed as `»map«` and aliased by the `clicks` value). This can be done by calling `clicks.AddHandler(hndl)`.

The state after adding the handler is displayed in **Figure 9** (B). There is a new link from the `»map«` event to the `hndl` value. This represents the fact that the event keeps a reference to the handler, so that it can trigger the handler. More interestingly, there are also new links from the event source along the entire event chain up to the `»map«` value. This is a result of the lazy initialization.

When we register the handler, the event checks whether the number of registered handlers was originally zero. If that's the case, it means that it hasn't yet registered an event handler for its source, which is the `»filter«` event in the diagram. The `»filter«` event performs the same check and possibly also registers event handler with its source. This way, the registration is propagated up to the primary source, which is an external event.

Thanks to the propagation, the call to `AddHandler` adds all the necessary *forward references*, so when the primary source event (`MouseDown`) occurs, all the event transformations will trigger and the provided handler will be called. Note that before adding the event handler to the constructed `clicks` event, the transformations wouldn't be called (and none of the functions we provided as an argument to `Event.map` and other combinators would run).

4.4 Removing handlers and losing references

There are several situations that can arise after we add an event handler. In this section, we'll analyze interesting combinations of removing the event handler and losing references to the chain.

Losing references. If the program returns from a routine that constructed the chain, it loses references to both the handler and the chain. As follows from our definition of garbage and the requirements (Section 4.1), we don't want to dispose the chain, because if the source event fires, it can trigger some useful handler.

We can see that our implementation behaves correctly in this case. Even if we lose the direct references to `»map«` and `hndl`, there are still references from the source to the handler. The links that keep the handler "alive" are forward references, which were created by propagating the call to `AddHandler`.

Removing handler. If we unregister the handler, but keep a reference from the program to the constructed event chain, the event checks whether the number of handlers reached zero (after the removal). If yes, it propagates the removal and removes its handler from its source event. Again, this may continue up to the original source event. This is a valid approach, because if there are no event handlers to trigger, we don't need to listen to the source.

The state of the event chain after adding and removing a single handler is exactly the same as the initial state after creation. In the diagram, we return back to the state in **Figure 9** (A). This means that the whole event chain is still in a usable state. As long as we keep a reference to the object representing the chain `»map«` we can again attach a handler and start listening to the event.

Removing handler and losing references. Now, let's look at the situation that motivated the work presented in this chapter. What if we remove the event handler and then lose references to the object representing the event chain (or remove handler after losing references)? In this case, the event chain becomes garbage. No one is listening to it, so it doesn't need to fire and we cannot attach an event handler to the chain, because we don't keep any reference to it. The original implementation of combinators is deficient in this case, because it keeps references from the event source to the end of the event chain, even though there are no attached handlers.

Our implementation using reversed links behaves differently. After removing the last handler, the removal is propagated, so there are no forward references in the chain. As shown in **Figure 9** (C), the only remaining references are reverse links from the end of the chain. We don't keep any reference to the chain, so all events that form the chain become garbage and can be collected.

The same situation occurs if the handler references the `»map«` value and unregisters itself, which was our original motivation. Until it does so, there are forward references, which keep the event chain alive, but once the last handler is removed, the removal is propagated and the chain is garbage collected.

4.5 Implementing sample combinator

In this section, we demonstrate our implementation of event combinators, by examining the `map` combinator. We already explained several aspects:

- Due to lazy initialization of event chains, no handler is registered with the source event when the combinator is called.

- When the first handler is added to an event constructed by the combinator, the combinator adds a handler to its source event.
- Similarly, when the last handler is removed, the combinator also unregisters itself from the source event.

The implementation directly follows these rules. It is more complicated than the version in section 1.2, but it can be simplified by composing combinators using primitive higher-order functions.

```
1: let map f (source:IEvent<_>) =
2:   let list = new ResizeArray<Handler<_>>()
3:   let this = new Handler(fun arg ->
4:     for h in list do h.Invoke(f arg))
5:   let add h =
7:     list.Add(h)
8:     if list.Count = 1 then
9:       source.AddHandler(this))
A:   let remove h =
B:     list.Remove(h)
C:     if list.Count = 0 then
D:       source.RemoveHandler(this))
E:   { AddHandler = add; RemoveHandler = remove }
```

The combinator first initializes a mutable list of registered handlers (line 2) and a handler (line 3) that will be later attached to the source event. The handler implements the projection, so when it is called, it iterates over all currently registered handlers and invokes them with the projected value as the argument (line 4).

On the next few lines, we define two local functions that will be called when the user adds a handler to the returned event (line 5) and when a handler is removed (line A). Both of the functions have similar structure. They first add or remove the handler given as an argument to or from the list attached handlers. Next, they check whether the change should cause propagation and register with or unregister from the source event (lines 9 and D).

4.6 Relations to standard GC techniques

At this point, it is worth discussing the relations between our implementation and standard garbage collection techniques.

Reference counting. As noted in section 1.1, our algorithm bears similarity to reference counting. A well-known problem of reference counting is that it fails to reclaim objects with reference cycles. We rely on GC implemented by the runtime when collecting objects or events that are not *object-reachable* and uses reference counting to collect events that are not *event-reachable*. This means that only cyclic references between events would be a problem. However, as noted in section 1.1, the set of combinators provided by F# library doesn't allow creating cyclic references between events, so the problem is avoided in our setting.

Weak references. Many runtime environments support advanced features for controlling the garbage collector, such as *weak references* and so it seems natural to ask whether these could be used in our implementation. We consider using weak references for forward or backward links, but as we'll see both of these uses would break the implementation.

If forward links were weak, the GC could collect events with attached handlers that perform some useful work (e.g. if we registered a handler with the event constructed in section 1.2 of Chapter III and then lost reference to the event chain). On the other hand, if backward links were weak, the GC could collect parts of a chain before registering the first handler (e.g. the `»filter«` event in **Figure 9** (A)). This discussion raises some interesting problems for future research.

5. Correspondence with the model

In the previous two sections, we've introduced a formal model and an implementation that is inspired by the model. In this section, we discuss the correspondence between the algorithm from section 5 and the library implementation.

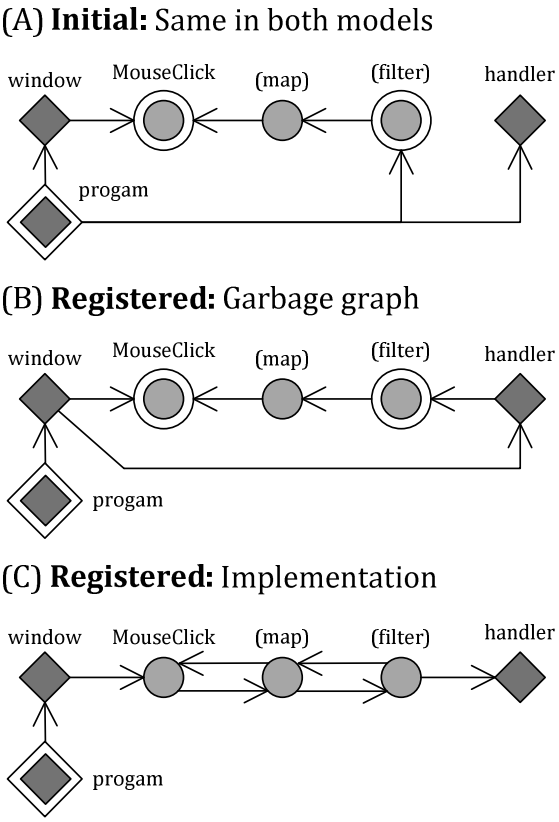


Figure 10. Formal algorithm and implementation. In the initial state, the implementation and the formal model give the same result (A). After registering a handler, the formal model (B) and the implementation (C) differ, but are equivalent in terms of reachability.

5.1 Duality of event references

When we create an event using a combinator, it only keeps a reference to previous event of the chain. This corresponds to the formal model, where we reverse all links leading from an event (7). If we look at a simple case, where none of the pre-processing steps take place, the formal model and implementation yield exactly the same result. An example is shown in **Figure 10** (A).

Once we attach an event handler to the constructed event, the situation becomes more complicated. We'll discuss this case in 5.2. The other pre-processing step is collecting unreferenced event values, which we'll discuss in section 5.3.

5.2 Mock references

When an event references a handler (object) in the formal model, we add a *mock reference* (6) from the event source (an object that references the first event of the chain) to the event handler (an object referenced by the last event). On the other hand, the implementation adds forward references by propagating the registration of the handler back to the source.

The Figure 10 demonstrates what happens in the formal model (B) as well as in the implementation (C). It shows the situation in which we're not directly referencing the event chain and the handler from the program (other cases would be similar). Even though the diagrams look different, it isn't difficult to observe that they are equivalent in terms of collectability of objects.

We argue that the events that are reachable in the formal model thanks to the mock reference added in the formal model (6) are the same as the events that are reachable in the implementation thanks to the forward references created thanks to the propagation of handler registration. We won't present a formal proof, but we can demonstrate this informally:

- The formal algorithm, adds a mock reference if there is a path visiting only events, from a source to a handler (both of them are objects). This mock reference corresponds to the fact that a handler was registered with the event. In the implementation, this causes a propagation of handler registration.

  (Note that when we reference an event from an object, the situation is different, because there is a link from the object to the event, but no link in the other direction.)

- As a result, we need to decide whether events, which become reachable thanks to mock references in the formal algorithm, correspond to the events that become reachable after registration propagation in the implementation.
- Let's take any event chain (event source, one or more events created by combinators and a handler). In the formal algorithm, all events and the handler are reachable, assuming that the source is reachable. This is the case, because there is a mock reference to the handler and reversed links from the handler up to the first event. In the implementation, events are also reachable, because references were added along the chain during registration propagation. The handler is also reachable, because the last event keeps a reference to it.

5.3 Event collection

The first pre-processing step in the formal model guarantees that we will collect all events that can't be triggered. In the implementation, there is no feature corresponding to this step. Let's demonstrate this mismatch using an example:

```
1: let test() =
2:   let form = new Form(Visible=true)
3:   form.MouseDown
4:     |> Event.filter (...)
5:     |> Event.map (...)
6: let evt = test()
7: evt.AddHandler(Handler(fun e -> printf "!"))
8: eventsList.Add(evt)
```

When we invoke the function in the code snippet (line 6), it constructs a visible form and returns an event. We add a handler to the event (line 7), and keep a reference to it in some global list of events (line 8). When the form is later closed, the runtime no longer keeps a reference to it, so the form as well as the constructed event chain should be collected (with the exception of the last event, which is directly referenced).

However, because of backward references, there is a link from the last event to all preceding events in the chain, so the entire event chain is kept alive. We don't find this limitation a problem in practice. In order to keep the event chain alive, the user needs to reference the constructed event value, but keeping a list of created events is rarely done in practice.

Similar situation arises in .NET and can be solved using the "Weak Delegate" pattern [21]. We provide combinator `Event.weak` [8] which allows the user to overcome this problem. This issue is, however, orthogonal to the one that motivated this chapter.

6. Chapter summary and related work

In this section, we review the related work in the area of garbage collection. We already discussed related programming models in Chapter II, so we only focus on the related work that is relevant only to this chapter.

6.1 Garbage collection research

When collecting events, we aim to collect objects and events that we consider as garbage, but that wouldn't be collected by the standard GC algorithm. This is related to research that provides the algorithm with an additional liveness property and collects not only objects that are *unreachable*, but also objects that are not *alive*. (e.g. [22, 23]).

The question whether we could collect objects matching the definitions in section 4 using similar techniques is an interesting future research problem that complements our work. One notable difference is that an event can be considered garbage even if it can still be triggered. As discussed in 3.3, this problem would have to be handled carefully in a GC algorithm implementation. Liveness analysis usually deals with objects that won't be accessed by the program, but the GC algorithm fails to recognize this.

6.2 Garbage collection of actors

The Actor model [16] describes concurrent programs in terms of active objects that communicate using messages, which is in many ways similar to the popular language Erlang [13]. Our imperative model is related to the actor model – waiting for an event plays the role of receiving a message and triggering an event corresponds to sending a message.

Actors face a similar problem, because garbage is defined in terms of the state of the actor (however, it cannot be expressed using the duality principle as in our work). Various specialized algorithms for collecting the garbage have been developed for the actor model [14]. However, the most relevant work is [15], which describes how to implement garbage collection for actors in terms of

standard garbage collection algorithm by translating the actor reference graph to a passive object reference graph. The specific steps performed by the algorithm are very different, but the general approach of manipulating the reference graph and using a standard GC algorithm is shared with our work (Section 5).

6.3 Conclusion

In this chapter, we discussed a problem with garbage collection of events, which emerges when we attempt to mix the declarative and imperative style and is also present in the current implementation of F# event combinators.

We defined the problem formally by providing a simple definition of *collectability* for reactive programming model. It combines both *objects* and *events* and benefits from the duality principle. Using this definition, we presented a garbage collection algorithm, which reclaims all *collectable* objects and events. The algorithm is based on graph transformation. This technique could be used to reduce the problem to well known GC algorithms.

However, our implementation aim wasn't to actually replace a garbage collection algorithm. Instead, we have shown an alternative implementation of library of F# event combinators, solely in terms of object references. Our implementation closely corresponds to the formal model and doesn't cause memory leaks when used in an environment that combines both declarative and imperative approach to reactive programming.

Chapter V

Pattern matching for reactive and concurrent programming

In this chapter, we show how to extend monadic computations to support pattern matching on monadic values and we look at several applications of this feature. The feature discussed in this chapter makes it possible to use pattern matching on events in the reactive library that we'll introduce later. However, the extension presented in this chapter is a general purpose language feature, so we won't focus our discussion on reactive programming. The key contributions presented in this chapter are the following:

- We show that a wide range of reactive and concurrent programming models can be encoded using a simple reusable language extension for monads that is based on pattern matching.

  Section 2 supports this claim by example. We show a reactive programming model (section 2.1) inspired by [17, 23]; a concurrent programming model (section 2.3) based on join calculus [31] bearing similarities to [32, 33]; and a parallel programming model (section 2.4) based on futures, which can express some features of Manticore [38].

- Our solution extends monads with two simple operations that are sufficient for encoding pattern matching on monadic values. We present a generalized pattern matching construct and show how it can be encoded in terms of those two operations (Section 3).

  We define laws that should hold about the two operations (Sections 4 and 5) and we observe interesting relationship with commutative monads and show that our syntactic extension is useful when working with them (Section 4.2).

- An essential aspect of our monadic pattern matching construct is that it preserves the usual intuition that users have about pattern matching in the ML-family of languages. We use the basic laws required for the two operations to prove numerous facts that are useful when reasoning about monadic pattern matching (Section 6).

We conclude this chapter by discussing several design alternatives that may be relevant for other languages or other computation types (Section 7). Our work is built on top of F# computation expressions, but the presented ideas should be applicable to any language with syntactic support for monads.

1. Motivation

We look at several examples that motivate the work presented in this chapter. We start by looking at an example from reactive programming, which we already briefly introduced in Chapter III. However, we'll show the problem in a somewhat different context. Then we'll show that pattern matching on monadic computations could also be useful when working with lists and finally, we'll look at the concurrent programming scenario.

1.1 Reactive programming

In Chapter III, we introduced an imperative programming model based on asynchronous workflows and we demonstrated the `Async.AwaitEvent` primitive, which can be used to wait for an occurrence of an event inside asynchronous computation. In this section, we won't use the overloaded version of `Async.AwaitEvent` for waiting on the first of multiple events, but we'll show how to encode similar thing using combinator library more explicitly.

The following example is, once again, taken from an application for drawing rectangles. After pushing the button, we enter a loop in which we need process two mouse events - mouse move, to update the rectangle, and release of the button, to finish drawing:

```
1: let rec drawing() = event {
2:   let! e = Event.combine w.MouseMove w.MouseUp |> Async.AwaitEvent
3:   match e with
4:   | Choice1Of2(m) -> redraw (m.X, m.Y)
5:                      return! drawing()
6:   | Choice2Of2(m) -> return m.X, m.Y }
```

The function `drawing` is called after the user presses the mouse button. It starts by waiting for either `MouseMove` or `MouseUp` event (line 2). To do this, we use a combinator `Event.combine` that creates an event which is triggered whenever any of the argument triggers. The combined event is passed to the `Async.AwaitEvent` function, which creates a workflow waiting for the first event occurrence. The combined event carries a discriminated union, so we can use pattern matching to determine which event happened. For `MouseMove`, we redraw the rectangle (line 4) and continue drawing (line 5). For `MouseUp` event, we return the coordinates of the end point (line 6).

The important point is that we need to leave the computation expression sublanguage when waiting for the first of two events. If we wrote two subsequent `let!` bindings, the code would first wait for the first event and then wait for the second one, so it would only continue after both of them would occur. Moreover, what if we wanted to add a third case, where we'd wait for `KeyDown` event, but only when the pressed key was `Esc` to cancel the drawing?

This seems like a perfect task for pattern matching, but it cannot be expressed directly, because we need actual values before we can use pattern matching. Currently, we'd have to use event combinators such as `Event.combine`.

1.2 Working with lists

Another frequently used type of computation is the well-known `List` monad. We can use it to model non-deterministic computations or for writing general list processing code. It isn't surprising that for some operations, we again need to leave the sub-language provided by computation expressions. To calculate non-zero differences between elements of two lists occurring at the same position, we need to use the `List.zip` function:

```
1: let calcDiffs xs ys = list {
2:   let! x, y = List.zip xs ys
3:   match x, y with
4:   | x, y when x <> y -> return y - x
5:   | _ -> return! [] }
6:
7: > calcDiffs [5; 5; 1] [5; 4; 8];;
8: val it : [-1; 7]
```

The function combines two input lists (line 2) and uses the combined list as an argument for the bind operator. This means that the pattern matching (line 3) will be executed for all pairs from the combined list. If the two values differ, we return a single result containing the difference (line 4). If the values are equal, we don't return any value (line 5).

The `List.zip` function is in some sense similar to the previous `Event.combine` combinator. It allows us to combine two monadic values (`List<'a>` or `Event<'a>` respectively) into a single one. The difference is that `Event.combine` gives us only a single unwrapped value at the time (either from the first or from the second monadic value) and `List.zip` gives us both values as a tuple.

In a way, `Event.combine` corresponds to pattern matching with two clauses, both of them with one value pattern and one underscore pattern (representing the fact that we don't need the value). On the other hand `List.zip` corresponds to pattern matching with only a single clause with two value patterns (meaning that we require both values). However, `List.zip` can't express the fact that a value is missing in one list, so it requires lists of equal lengths. If we for example want to pad a shorter list with values from a second list starting at the index where the shorter list ends, we have to write a recursive processing function:

```
let rec padWith vals defs =
  match vals, defs with
  | v::vs, _::ds -> v::(padWith vs ds)
  | _, ds -> ds

> padWith [ 11; 12; 13 ] [ 1 .. 5 ];;
val it : [11; 12; 13; 4; 5]
```

The underscore pattern provides a natural way for expressing that a value is not available, however this cannot be written in the language provided by computation expressions.

1.3 Concurrent programming

Our last motivation is based on Join calculus [31], which provides a declarative way for expressing synchronization patterns. Joins have been used as a basis for

language features [32, 33], but it is also possible to implement them as a library [34, 35]. Scala provides an elegant integration thanks to extensible pattern matching [36]. The following example shows a simple unbounded FIFO buffer implemented in Scala. It uses two channels and consists of a single join pattern:

```
1: val Put = new AsyncEvent[Int]
2: val Get = new SyncEvent[Int]
3:
4: join { case Get() & Put(num) =>
5:          Get reply num }
```

The example first declares two channels (called events in Scala). The first one (line 1) is asynchronous, which means that it doesn't block the caller when invoked. When called, it stores an integer value in a buffer and returns immediately. The second one (line 2) is blocking. When it is invoked, it blocks the caller until a value (provided by a call to `Put`) is available. If there was a previous call to `Put`, it takes the value from the buffer and returns immediately.

The join pattern that implements this behavior is encoded using pattern matching (line 4). It specifies that the body (line 5) should be called when there is a value in the `Put` buffer and when there is a pending call to `Get`. When that's the case, the body is called and it returns a value `num` to the caller of the `Get` channel.

In F#, we can embed join patterns into computations based on asynchronous workflows. This has an important benefit. Asynchronous workflows don't block threads while waiting, so we can avoid creating an unnecessary number of expensive threads. In the following snippet, the bind operation represents waiting for a first value from the channel:

```
1: let put = new Channel<int>()
2: let get = new Channel<ReplyChannel<int>>()
3:
4: let rec buffer() = join {
5:   let! x = put
6:   let! chnl = get
7:   chnl.Reply(value)
8:   return! buffer() }
```

We start by defining two channels just like in the previous Scala version. Inside the join computation, we first wait for a number from the `put` channel (line 5). Next, we wait for a value from the `get` channel (line 6), which give us a *reply channel*, which can be used for returning the result (line 7).

In this simple case, using `let!` to represent waiting for a value works, because we need to wait for both of the values in every case. Also, the order of waiting doesn't matter, because values are buffered. However, in general, we need more expressive power. The following Scala example allows us to put two different types of values into the buffer:

```
1: join {
2:   case Get() & PutInt(x) =>
3:     Get reply ("Number: " ++ x.toString())
4:   case Get() & PutString(x) =>
5:     Get reply ("String: " ++ x) }
```

In this example, we have two different `Put` channels. The first one is used for storing integer values in the buffer, while the second one stores string values. When `Get` is called, it waits for the first value from any of the two channels. In the implementation, we use two *join patterns*. The first one (line 2) is triggered when there is a value in `PutInt` and a pending call to `Get`. The second one (line 4) is similar, but takes a value from `PutString`.

Encoding this example using our previous `join` computation is tricky. We need to wait for `Get` in any case (using `let!`), but we cannot express waiting for either `PutInt` or `PutString`, without declaring some combinators. This would get even complicated if we wanted to provide features such as pattern matching on values inside the channel. Once again, it seems that pattern matching on monadic value would solve the problem far more elegantly.

2. Monadic pattern matching

In this section, we present an overview of our pattern matching on monadic computations. As we'll see later, a fully general pattern matching requires providing two operations in addition to *bind* and *return*. We'll introduce them gradually, first looking at examples that require only one of them.

2.1 Merging computations

Pattern matching is very often used on tuples. If we want to use pattern matching on monadic computations, we need some way for merging two computations into a single one containing tuples. To enable this, we require the *merge* operation:

```
val ⓜ : M<'a> -> M<'b> -> M<'a * 'b>
```

Judging just from the type, it should be possible to implement *merge* in terms of *bind* and *return*. For some monads, this may be the right choice, but not for all of them. We'll discuss the *merge* operation in details in section 4 and focus on an example for now. When merging two lists in section 1.2, we used `List.zip`, which has the same type signature as our *merge*. We can define `list` computation builder that uses `List.zip` as *merge*. Using the operation explicitly we can write for example the following code:

```
let numbers xs ys = list {
  let! xys = xs ⓜ ys
  match xys with
  | x, y -> return 10 * x + y }

> numbers [1; 2; 3] [6; 5; 4];;
val it : int list = [16; 25; 34]
```

This example is the same as the code we would write when using `List.zip` explicitly. The reason why we're showing it is that the same thing can be written using our monadic pattern matching construct. The following function means exactly the same thing:

```
1: let numbers xs ys = list {
2:   match! xs, ys with
3:   | !x, !y -> return 10 * x + y }
```

The `match!` construct takes one or more monadic values as arguments (line 2). In our example, we provide two values of type `list<int>`. The patterns we specify on line 3 are special syntactic construct that we call *computation patterns*. In the example, we use two computation patterns of the form `!pat`, where `pat` is a usual F# pattern (we call this form a *binding pattern*). It specifies that we need to get an actual value from the monad and the value should match the pattern `pat`. We'll explore the second form of computation pattern in the next section.

The previous example can be defined in terms of the *merge* operator, because it is relatively limited. It contains only a single clause and both patterns for values are complete, meaning that they will match any given value. For any more complicated use we'll need our second operation.

2.2 Choosing a computation

Pattern matching with a single clause isn't all that useful. To support multiple clauses in pattern matching on monadic computations, we need an operation that allows us to select among multiple clauses. This isn't as straightforward, because we need to do it "inside a monad", which can behave in many diverse ways. The operation that we use for encoding multiple clauses is called *choose*. The following type signature is slightly simplified, because it ignores the possibility that a pattern may fail, but we'll get to the fully general case shortly:

```
val choose : list<M<M<'a>>> -> M<'a>
```

It takes a list of computations as an argument. Each of the computations in the list carries a monadic computation as a value. This wrapped computation is a computation that should be called when the clause is selected. Interestingly, the *choose* operation looks quite similar to *join*, except it takes a list of `M<M<'a>` computations instead of just a single one and, indeed, it is a generalization of *join*. We'll talk about *choose* in details in section 4 after looking at several other examples.

In the section 1.1, we've seen an example where we wanted to wait for one of several events depending on whichever occurred first. We can encode this behavior using our new `choose` function. We need to provide a list of computations of type `Event<Event<int * int>>`, which can be done using `map`. For each of the events, we project the carried value into an event computation that should be executed if it is chosen:

```
1: let rec drawing() =
2:   choose [
3:     map (fun m -> event {
4:       redraw (m.X, m.Y)
5:       return! drawing() }) (await w.MouseMove);
6:     map (fun m -> event {
7:       return m.X, m.Y }) (await w.MouseUp) ]
```

Just like in section 1.1, the `drawing` function returns a single monadic computation of type `Event<int * int>`. We construct it by creating two asynchronous workflows that wait for `MouseMove` and `MouseUp` events respectively (lines 3, 6) and then selecting the computation that first has a value using `choose`. Each of the two cases returns another computation (written using the `event` builder), which specifies

code to run in case it is selected. This is either redrawing the window and looping (lines 4, 5) or returning the last mouse position (line 7).

As already mentioned, the *choose* operation is used when we want to write pattern matching on monadic computations with multiple clauses. Let's rewrite the previous example using our syntactic sugar. We'll use the two asynchronous workflows as arguments of `match!` and two clauses, each of them matching any value carried by the first, respectively the second event:

```
1: let rec drawing() = async {
2:   match! w.MouseMove |> Async.AwaitEvent,
3:          w.MouseUp  |> Async.AwaitEvent with
4:   | !m, _ -> redraw (m.X, m.Y)
5:              return! drawing()
6:   | _, !m -> return m.X, m.Y }
```

As already explained in section 2.1, the clauses of `match!` are formed by *computation patterns*, which is a special syntactic category. The form `!<pat>` means that we need to obtain a value from the monadic computation (in this case, wait until an asynchronous workflow completes), while the *ignore pattern* (written as underscore) means that we don't need a value. Note that there is a difference between "_" and "!_". In the first case, we don't need the value at all, while in the second case, we need to obtain the value (i.e. wait for an event), but ignore it afterwards.

As we can see by analyzing the patterns, each of the clauses waits only for one of the two events (monadic computations), which is why we didn't need the *merge* operation in this example.

2.3 Merging and choosing together

In a general case, we have multiple clauses, each of them having multiple *binding patterns*. To demonstrate this, we get back to concurrent programming and our example motivated by joins. In section 1.3, we've seen a Scala example with two channels for putting values (integers or strings) into a buffer and a single *get* channel. The buffer was implemented using two joins that combined the *get* channel with the first or the second *put* channel. We can encode the same idea using `match!` like this:

```
1: let putInt = new Channel<int>()
2: let putString = new Channel<string>()
3: let get = new Channel<ReplyChannel<string>>()
4:
5: let rec buffer() = join {
6:   match! get, putInt, putString with
7:   | !chnl, !n, _ ->
8:       chnl.Reply("Number: " + n.ToString())
9:   | !chnl, _, !s ->
A:       chnl.Reply("String:" + s) }
B:   return! buffer() }
```

As we can see, each clause combines two channels (lines 7 and 9) and ignores the third one. If we get an integer value and a reply channel `chnl` in the first join pattern (line 7), we send a number converted to a string as the reply (line 8). The second join pattern is quite similar. After we process one pair of messages, we recursively wait for another join (line B). Unlike for example in Cω, this is written

explicitly, because we may want to continue by handling another combination of join patterns (encoded as another `match!` computation).

Note that this example isn't a standard monadic computation. The type of values that we're using as parameters to *bind* (or our more general *choose* operation) is `Channel<'a>`, while the type of the constructed computation is `Async<unit>` (an asynchronous workflow, which was introduced in section 2.1). Since F# computation expressions including our extension are handled as a simple syntactic transformation, it is possible to use operations of atypical type signatures. Even though this is a non-standard example, it is important because it shows the relationship of our extension with previous related work. Moreover, it is also very useful in practice.

2.4 Choosing a computation with failures

The last feature of our `match!` construct that we haven't introduced yet is that we can use more complicated F# patterns inside *binding patterns* of clauses, including patterns that can fail. In that case, the behavior depends on the author of the computation. Typically, it may try all clauses and then throw an exception if no matching clause is found or wait if the computation can produce a different value later and retry the patterns. However, using the *choose* operation as we introduced it earlier, there is no way to represent match failure.

Let's start by looking at an example. In this case, we'll work with `future` builder, which creates computations of type `Future<'a>` (inspired by Manticore). It represents a computation that is (or may be) running in the background and eventually produces a value of type `'a`. We can use the `match!` construct inside this computation to write a function that multiplies values of all leaves of a binary tree in parallel:

```
1: let rec treeProd t = future {
2:   match t with
3:   | Node(lt, rt) ->
4:       match! treeProd lt, treeProd rt with
5:       | !0, _ -> return 0
6:       | _, !0 -> return 0
7:       | !a, !b -> return a * b
8:   | Leaf n -> return n }
```

The function starts by standard pattern matching (line 2). If the tree is a node with left and right branch, it recursively calls itself to creates two futures to process both of the branches (line 4). Next, we need to wait for both of the futures to produce a value, which is done using monadic pattern matching with two *binding patterns* (line 7). In case that one future completes earlier and produces 0, we know the overall result immediately, so we included two clauses to handle this special case (lines 5 and 6).

When we use `match!` with futures, it waits for the first future to produce a value and then checks whether it can run any of the clauses. If yes, it follows the selected clause and cancels other futures. In the other case, it waits for more futures to complete.

Earlier we said that *choose* takes a list of computations that contain computations to be used if the clause is selected. This wasn't exactly correct. The outer computation may report that the pattern matching failed or that it succeeded and produced an (inner) computation that can be used to continue with. The actual type signature of *choose* is following:

```
type MaybeDelayed<'a> =
  | Success of (unit -> 'a)
  | Failure

val choose : list<M<MaybeDelayed<M<'a>>>> -> M<'a>
```

When compared with the signature shown earlier, the only change is that the inner `M<'a>` computation is now wrapped inside `MaybeDelayed<'a>`, which allows us to represent pattern matching failure. In F#, we can write monadic computations that are not lazy and contain side-effects, so the wrapping also guarantees that we won't evaluate the body of clause before it is actually selected.

We look at the details of the syntactic transformation that the F# compiler performs in section 3. However, let's at least briefly look at the code produced for the two clauses on lines 6 and 7. In the following listing, the values `f1` and `f2` store the result of calling `treeProd` on `lt` and `rt` respectively:

```
1: choose [
2:   ...
3:   map (function
4:    | 0 -> Success(fun () -> future { return 0 })
5:    | _ -> Failure) f2;
6:   map (function
7:    | a, b -> Success(fun () -> future {
8:        return a * b })) (merge f1 f2) ]
```

The first clause is translated into a computation that applies the `map` operation to the `f2` value (lines 3-5). The function given as an argument to `map` gets a value of the future, which is an integer. If the value is 0, it returns `Success` with a future computation to run (line 4) otherwise it returns `Failure` (line 5). The second clause is similar, with the exception that it first combines two futures using `merge` and the pattern matching always succeeds.

The interesting case is when `f2` produces a value. As a result, the first computation of the list we gave to `choose` also finishes. If it produces `Success`, the `choose` operation cancels all other futures in the list (which in turn cancels the `f1` future), evaluates the function stored inside `Success` and runs the provided body. In case of non-zero result, it continues waiting until some other clause produces `Success`. If all clauses produce `Failure`, then the `choose` operation throws a match failure exception.

Our implementation of `match!` for futures is similar to the `pcase` (parallel case) construct known from Manticore [38]. The parallelism in Manticore is implicit. However, we achieve similar syntactic simplicity and expressive power just by using a generally useful language feature.

cpat = _	Computation ignore pattern
| **!***pat*	Computation binding pattern
ccl = $cpat_1$, …, $cpat_k$ -> *cexpr*	Computation match clause
| $cpat_1$, …, $cpat_k$ **when** *expr* -> *cexpr*	Computation match clause with guard condition
cexpr = **match!** $expr_1$, …, $expr_k$ **with** ccl_1 | … ccl_l	Pattern matching computation (sequence of computation clauses)
| …	Other computation expression

Figure 11. Syntax of monadic pattern matching

3. Semantics

In this section, we discuss the semantics of our extension. We present syntax and a translation to the core language. We follow the design of computation expressions [28] and active patterns [39] and do not give typing rules explicitly. The types are checked after the translation using standard F# type-checking rules. The second important part of the semantics is the laws that we require to hold about `merge` and `choose` operation. These will be discussed later in sections 5 and 6.

3.1 Syntax

The syntax of our extension is shown in Figure 11. In addition the standard constructs described in section 3.3 of Chapter II, we add a single new case to the *cexpr* category. The `match!` construct takes one or more expressions as arguments and has one or more clauses.

Clauses do not consist of standard *patterns*, but are formed by *computation patterns*, so we need to introduce a new syntactic category for clauses (*ccl*) and a new category for computation patterns (*cpat*). A *computation clause* looks like an ordinary clause with the exception that it consists of *computation patterns* (instead of usual *patterns*) and the body is *computation expression* (instead of standard *expression*).

Finally, a computation pattern can be either an *ignore pattern* (written as "_") or a *binding pattern*, which is a standard F# pattern [30] prefixed with "!". We've already seen several computation patterns when discussing the examples. For example we can write `!0`, which is a binding pattern constructed from a constant pattern that matches an integer against a zero. Notably, we also support active patterns [39] in *computation patterns*. In the reactive programming example, we could define a pattern `LeftClick`, which succeeds only when an event is a left button click. Then we could use the computation pattern `!LeftClick`.

3.2 Translation

We describe a translation that transforms pattern matching computation into an ordinary expression that doesn't contain computation expressions. We extend the translation described in Figure 5 by adding the case for the `match!` construct and we also define rules for translating computation clauses. The **Figure 12** shows a rule (1) which translates a computation expression into an ordinary expression. The rest of the rules define the following two translation functions:

$[\![-]\!]_{cexpr} : cexpr \times ident \to expr$

$[\![-]\!]_{ccl} : ccl \times ident \times [ident] \to expr$

The first function takes a *computation expression* and an identifier representing the computation builder of the monad. The second function also takes a list of identifiers, which we use to pass arguments of `match!` to the function for translating clauses.

$$[\![expr \{ cexpr \}]\!] = \textbf{let } m = expr \textbf{ in } [\![cexpr]\!]_{cexpr}\ m \quad (1)$$

$$[\![cpat_1, \ldots, cpat_k \text{ -> } cexpr]\!]_{ccl}\ m, (v_1, \ldots, v_k) = \quad (2)$$
$$\text{map}_m\ (\textbf{function}$$
$$\quad | (pat_1, \ldots), pat_n \text{ -> Success(}\textbf{fun}\ () \text{ -> } [\![cexpr]\!]_{cexpr}\ m)$$
$$\quad | \_ \text{ -> Failure })\ cargs$$
$$\text{where } \{ (pat_1, v_1), \ldots, (pat_n, v_n) \} = \{ (pat, v_i) \mid cpat_i = !pat;\ 1 \le i \le k \} \quad (3)$$
$$\text{and } cargs = v_1 \mathbin{⩒_m} \ldots \mathbin{⩒_m} v_{n-1} \mathbin{⩒_m} v_n \quad \text{for } n \ge 1 \quad (4)$$

$$[\![\textbf{match!}\ expr_1, \ldots, expr_k \textbf{ with } ccl_1 \mid \ldots ccl_l]\!]_{cexpr}\ m = \quad (5)$$
$$\textbf{let } v_1 = expr_1 \textbf{ in } \ldots \textbf{ let } v_k = expr_k \textbf{ in}$$
$$\text{choose}_m\ [\ [\![ccl_1]\!]_{ccl}\ m, (v_1, \ldots, v_k); \ldots ; [\![ccl_l]\!]_{ccl}\ m, (v_1, \ldots, v_k)\ ]$$

Figure 12. Translation of monadic match

In the translation rules, $bind_m$ denotes the `bind` operation implemented by a computation builder instance, which is stored in the value named `m`. In reality, the F# compiler expects that operations are provided as members and accesses them using the dot-notation (for example `m.Bind`). When translating `match!` (5), we first construct a new value for each of the arguments. This guarantees that any side-effects of the expressions used as arguments will be executed only once. The rest of the rule translates all clauses of the pattern matching and creates an expression that chooses one clause using the $choose_m$ operation.

Translation of a clause is slightly more complicated (2). It needs to identify which of the `match!` arguments are matched against the *binding pattern*. This is done in (3) where we construct a list containing an ordinary pattern (extracted from the binding pattern) and a monadic value, which should be matched against it. Next we combine all monadic values that are needed in the clause into a single value using the merge operator $⩒_m$ (4). The operator is left-associative, so when combining for example three values of types `'a`, `'b`, and `'c`, the resulting value will be of type `M<('a * 'b) * 'c>`.

Finally, we pass the combined monadic value as an argument to a map_m operation. It "extracts" the actual value of the monadic computation, runs the provided projection and then again "wraps" the value. In the projection function, we match the actual value against the patterns extracted earlier. If the matching succeeds we return `Success` containing a delayed and translated body of the clause. The result of translating a *computation clause* will be of type `M<MaybeDelayed<'a>>`.

The translation imposes one restriction that isn't reflected in the syntax. In particular, when translating a clause, we require that the clause contains at least one binding pattern (4), which ensures that *cargs* will be initialized to some mona-

dic value. Allowing this case would require a special handling and it is not clear what the semantics of this clause should be in general. We discuss this problem further in section 7.

4. Merging computations

We've seen numerous examples of using the monadic pattern matching in practice and we've introduced the two operations that are used to encode the `match!` construct. In the next two sections, we'll focus on these two operations in details, starting with *merge*.

4.1 Merge operation laws

As already discussed, *merge* generalizes the zip function for lists and similar operations that appear in reactive and concurrent programming. The operation should have the following type:

```
val ⫼ : M<'a> -> M<'b> -> M<'a * 'b>
```

We also require several laws to hold about it. The laws ensure that the operation behaves intuitively and allow us to guarantee some properties of our `match!` construct, which will be discussed later:

let assoc $((a, b), c) = (a, (b, c))$
let swap $(a, b) = (a, b)$

$$\begin{aligned} \text{return } (a, b) &\equiv (\text{return } a) \mathbin{⫼} (\text{return } b) && \text{(C1)} \\ u \mathbin{⫼} (v \mathbin{⫼} w) &\equiv \text{map assoc } ((u \mathbin{⫼} v) \mathbin{⫼} w) && \text{(C2)} \\ u \mathbin{⫼} v &\equiv \text{map swap } (v \mathbin{⫼} u) && \text{(C3)} \end{aligned}$$

We formulate the laws using *map* (instead of *bind*) to make them more intuitive. The first law (C1) specifies how the *merge* operation should behave with respect to *return*. It may be of interest that this law is very similar to the *product law* of causal commutative arrows [49], which relates parallel composition operator `(***)` in place of our ⫼ with an *init* function (in place of *return*).

The next two laws resemble associativity (C2) and commutativity (C3). The law (C3) is particularly interesting. It for example forbids using Cartesian product of two lists as the *merge* operation of a list monad. Even though the elements of the two returned lists would be the same, their order in the lists would be different! However, the rule is important because it guarantees that rearranging the order of `match!` arguments (together with rearrangements of clause patterns) does not change the meaning of program. We discuss the possibility of using Cartesian product semantics when working with lists in details in section 4.3.

4.2 Merging in commutative monads

The law resembling commutativity also reveals interesting relations between our *merge* operation and the *bind* operation in commutative monads. When introducing *merge* in section 2.1, we noted that by looking at the type signature, it seems possible to implement it in terms of *bind* and *return*. Now that we've also specified what laws should hold about *merge*, we can finally complete this idea.

Implementing merge. It appears that we can implement *merge* by applying the *bind* operation on both of the *merge* arguments (in a sequence) and then combining the obtained value into a tuple. Using the computation expression syntax:

```
let ⓜ (ma:M<'a>) (mb:M<'b>) : M<'a * 'b> =
  m { let! a = ma
      let! b = mb
      return a, b }
```

This operator has the right type, but if we analyze whether it obeys all laws, we find a problem. The law (C3) says that changing the order of *merge* arguments should only change the order of elements stored in the returned tuple. But that's not necessarily the case here.

For lists, the previous implementation behaves as Cartesian product. As discussed earlier, cross-product breaks the (C3) law, because changing the order of arguments changes the order of generated elements. To give a second example, in our earlier reactive programming monad, binding means waiting for the first occurrence of an event. When waiting in sequence, `mb` can occur several times before the `ma` occurs for the first time, so we would get 1st value of `ma` and for example 3rd value of `mb`. Waiting in the reversed order can give completely different results. We cannot use this straightforward implementation blindly. However, we'll see that it obeys the *merge* laws in one important special case.

Commutative monads. In the Chapter II, we discussed the three monad laws that should hold about any monadic computation. Additionally, a monad is *commutative* if it obeys one more law. It specifies that the order of binding doesn't matter (specifically, when using monads to control effects, it means that the order of effects makes no difference). For a commutative monad, the following two expressions are equivalent[8]:

```
m { let! a = <expr>1        m { let! b = <expr>2
    let! b = <expr>2   ≡        let! a = <expr>1
    <cexpr> }                   <cexpr> }
```

(Assuming that *a* and *b* do not appear in *<expr>1* and *<expr>2*)

Examples of commutative monads that are often used in practice include `Reader` (used for reading values from an environment), `Maybe` (representing computations that may fail) or `RandomMonad` (computations that use random numbers). In his Haskell retrospective [41], Simon Peyton Jones included commutative monads as one of open challenges. Even though the order of bindings doesn't matter, we can work with them only using the usual, overly sequential, notation.

Implementing merge (again!) For us, commutative monads are important, because the above implementation of *merge* in terms of *bind* and *return* is correct for any commutative monad. Using the commutative law, we can easily verify that the following two declarations are equivalent:

[8] F# doesn't use monads to control side-effects, so the expressions *<expr>1* and *<expr>2* may contain side-effects, which would be indeed reversed. The equivalence focuses only on effects that are controlled by the monad.

```
let ⫴ ma mb = m {          let ⫴ ma mb = m {
  let! a = ma                 let! b = mb
  let! b = mb          ≡      let! a = ma
  return a, b }               return a, b }
```

For commutative monads, the above code gives a correct definition of the *merge* operation (a proof is presented in Appendix C). To verify the (C1) law, we can substitute the above implementation of merge operation for the ⫴ operator. Applying the left identity law to the right hand side twice gives us the left hand side. Verifying (C2) requires more steps. We use associativity and left identity to move the application of *assoc* to the inner-most expression, expand it and then reduce it to the expression on the left-hand side.

The most interesting law is (C3). To prove that it holds for the above definition, we use *associativity* and *left identity* as in the previous case to move the application of *swap* to the inner-most part of the expression (similarly as in the previous case). However, then we need to use the additional law of commutative monads to prove the two expressions equivalent. For monads that are not commutative, this wouldn't be possible.

Pattern matching syntax. When working with commutative monads just using *bind* and *return,* we're forced to use a sequential notation (as noted in [41]). Our generalized pattern matching offers a simpler alternative. For example, suppose that we're working with computations that can fail. We can use the commutative `Maybe` monad to write such code. Let's say that we have four values that represent a rectangle location (`mleft`, `mtop` and `mwid`, `mhgt`). We want to calculate the center of the rectangle. If the computations couldn't fail, we could simply write:

```
(mleft + mwid/2, mtop + mhgt/2)
```

Unfortunately, inside monadic computation, we first need to extract the actual values using four bindings (using `let!`) that are written in a sequence:

```
maybe { let! l = mleft
        let! w = mwid
        let! t = mtop
        let! h = mhgt
        return (l + w/2, t + h/2) }
```

The *merge* operation for `Maybe` can be defined in terms of *bind*. Then we can use `match!` to write the code as follows:

```
maybe { match! mleft, mtop, mwid, mhgt with
        | !l, !t, !w, !h ->
            return (l + w/2), (t + h/2) }
```

The code is still more complicated than the non-monadic version. However, we can obtain values of all parameters using syntax that doesn't unnecessarily sequentialize the code. Even though providing an elegant syntax for working with commutative monads isn't the goal of this paper, we can see that our generalized pattern matching construct is certainly interesting from this point of view as well.

4.3 Cartesian product and Zip semantics

When designing syntax for working with lists, it is often a question whether a combination of multiple lists should behave as the zip operation or as a Cartesian product. For example, Haskell list comprehensions [43] use Cartesian product semantics, but Data Parallel Haskell [44] uses zip semantics.

As discussed in section 4.1, we need to use the zip semantics for our *merge* operation on lists. Cartesian product doesn't obey the commutativity law (C3), which means that we can't reorder the parameters of the `match!` construct. The following example demonstrates a simple list processing function that is implemented using `match!` with the zip semantics.

```
let numbers xs ys = list {
    match! xs, ys with
    | !x, !y -> return 10 * x + y }

> numbers [1; 2] [5; 6]
val it : list<int> = [15; 26]
```

Clearly, the Cartesian product semantics is also very useful, so we can ask whether there is any way to define a *merge* operation with Cartesian product semantics that would obey the laws from Section 5.1. Originally, the commutativity law (C3) didn't hold, because changing order of arguments reorders the elements of the generated list. We can overcome this problem by working with a data structure that isn't ordered, such a bag (also called multiset). The following example demonstrates working with a bag:

```
let numbers xs ys = bag {
    match! xs, ys with
    | !x, !y -> return 10 * x + y }

> numbers (bag [1; 2]) (bag [5; 6])
val it : bag<int> = bag [15; 16; 25; 26]
```

Note that in this example, the commutativity law (C3) holds because the following equation is true:

```
bag [15; 16; 25; 26] ≡ bag [15; 25; 16; 26]
```

Both of the options are useful in practice and the distinction between ordered lists and unordered bags makes it possible to choose between them depending on the user's current needs.

4.4 Merging and applicative functors

Applicative functors (also called *idioms*) [42] are another form of computations related to monads. Applicative functors are weaker than monads. This means that every monad defines an applicative functor, but not all applicative functors are also monads. An applicative functor is defined in terms of two operations:

```
pure : 'a -> F<'a>
⊛   : F<'a -> 'b> -> F<'a> -> F<'b>
```

However, there is an alternative (but equivalent) way to define applicative functors using the following three operations:

```
unit : F<unit>
map : ('a -> 'b) -> F<'a> -> F<'b>
⋆  : F<'a> -> F<'b> -> F<'a * 'b>
```

The signature of the ⋆ operation should look very familiar. In fact, it has exactly the same type as our *merge*. Even though the types are the same, the operations are different, because the set of laws that must hold about them differs. The following laws should hold about any applicative functor:

$$\begin{aligned} \text{map assoc } (u \star (v \star w)) &\equiv (u \star v) \star w && \text{(Associativity)} \\ \text{map } (f \times g)\ (u \star v) &\equiv \text{map } f\ u \star \text{map } g\ v && \text{(Naturality)} \\ \text{map snd } (\text{unit} \star v) &\equiv v && \text{(Left identity)} \\ \text{map fst } (u \star \text{unit}) &\equiv u && \text{(Right identity)} \end{aligned}$$

The ⋆ operation must obey the *associativity law*, which we also required for ⫴. The other laws are different and are not equivalent. As discussed in [42], any monad is also an applicative functor (meaning that it defines ⋆), but as we said in Section 5.2, not all monads can automatically provide ⫴. This is because we also require *commutativity* law (C3), which follows neither from the monad laws, nor from the applicative functor laws.

```
let ⊛ fs xs = m {
  let! f = fs
  let! x = xs
  return f x }
```

```
let ⊛ fs xs = m {
  match! fs, xs with
  | !f, !x -> return f x }
```

Figure 13. Defining applicative functors

Defining application. For any monad, we can implement the ⊛ operation using *bind*. This is always a valid definition and it is shown in **Figure 13** (left). If we have a monad with the *merge* operation, we can implement the ⊛ operation in terms of *merge*. **Figure 13** (right) shows how to do this using the `match!` construct. If we translate the function to underlying function calls and simplify it (by removing *choose* operation, which doesn't have any effect in this case) we get the following:

```
let ⊛ fs xs = map (fun (f, x) -> f x) (fs ⫴ xs)
```

Unsurprisingly, this declaration is the same as the one used in [42] to define ⊛ in terms of ⋆ when showing that the two formulations of applicative functors are equivalent. This way of defining ⊛ yields a function with a correct type, but it doesn't guarantee that the laws of applicative functors will hold. It may give a valid (and useful) version of applicative functor, but we need to check the laws when using it.

Choosing application. The question we now face is which of the two implementations should we use? For commutative monads that use the default definition of *merge* (from Section 5.2), the answer is simple – the two definitions are equivalent.

For other monads, the situation is more complicated. In our example with lists from Section 3.1, we started with a list monad. Its *return* operation returns a singleton list and *bind* performs projection followed by concatenation. Our *merge* operation was implemented as `zip`. In this case we cannot define ⊛ using *merge*, because we wouldn't get a correct applicative functor. In the last law, the left-hand

side zips some list with a singleton list, which always produces a singleton list. However, the right hand side could be a list of arbitrary length.

Applicative functor based on `zip` is a classic example, but the problem is that it needs a different *return*. The *return* operation should produce an infinite list containing the specified value repeatedly. This doesn't mean that we cannot use `match!` to solve problems that can be solved by applicative functors. As we'll see shortly, it just cannot do done fully automatically. For some other monads, the definition of ⊛ in terms of *merge* gives an alternative applicative functor that can be also useful.

We'll look at Imperative streams [3], which motivated our reactive programming examples, but are defined as a pure monad. A stream has discrete time. The *bind* operation extracts a value at the current time. As a result, when using *bind* to write ⊛, we get an operation with zip-semantics. However, we can provide *merge* operation which behaves as a cross-product (with a well-defined ordering, so that the laws from Section 5.1 hold). Then we can use it to define an alternative useful instance of applicative functor.

Encoding applicative examples. Even though we cannot always use the *merge* operation to define an applicative functor, we can still use it to implement some problems that are nicely solved using applicative functors. One useful applicative functor for lists has `zip` as the ⊛ operation and a function that generates an infinite list containing the specified value as *pure*. It can be, for example, used to define a transposition of matrix represented as a list of lists. The Haskell implementation looks as follows:

```
1: trans :: [[a]] -> [[a]]
2: trans    []    = pure []
3: trans (xs:xss) = pure (:) ⊛ xs ⊛ trans xss
```

When the input is an empty list (line 2) we return a lazily generated infinite list containing empty lists. For a non-empty list (line 3) we get a list `xs` containing the first row and a list `xss` containing the remaining rows. Using applicative operations, we apply the *cons* operation to all elements of the first row and rows of the transposed remainder. We can write the code using our `match!` extension (behaving as `zip`) by following a similar pattern:

```
1: let rec trans m = lazyList {
2:   match m with
3:   | LazyNil -> return! repeat LazyNil
4:   | LazyCons(xs, xss) -> match! xs, trans xss with
5:                          | !y, !ys -> return LazyCons(y, ys)
```

When the matrix is an empty list (line 3), we need to generate a possibly infinite list of empty lists. This cannot be done using `return` primitive, because that is the monadic *return* (yielding a singleton list). Instead we generate the result using the `repeat` function. Note that `return!` in a computation expression returns the given list as it is. In the second case (line 4), we recursively transpose the remainder of the matrix and "zip" the result with elements of the first row (line 4). Then we apply the *cons* operator to all of the pairs and return the composed list as the next row of the transposed matrix (line 5).

Even though the structure of the two examples isn't exactly the same, there are similarities. If we expanded the use of ⊛ in the first version to the definition using `match!` shown in **Figure 13** (right) the code would start looking alike. In general, this example shows that we can often use the `match!` syntax to solve problems that would be otherwise solved by applicative functors that are not monads (such as list with zipping semantics of the ⊛ operation). This gives F# computation expressions with our extension an additional and very useful expressive power.

5. Choosing

We've introduced the choose operation gradually in sections 3.2 (where patterns always succeeded) and 3.4 (where we added support for failures). The fully general version of the function takes a list of monadic computations that yield the results of pattern matching. A result may be `Failure` if the pattern matching fails or `Success`, which carries delayed body of the selected clause (see also Section 3.4):

```
val choose : list<M<MaybeDelayed<M<'a>> -> M<'a>
```

The signature resembles the type of monadic *join*. It is extended to support failures and choose one of multiple computations. This is not accidental and as we'll discuss in section 6.2, the *choose* operation should be a generalization of *join*.

The explicit representation of failures makes the type signature more complicated. However, as we'll see in section 8.1, the additional complexity is needed if we want to follow the usual behavior of ML-style of pattern matching.

5.1 Choose operation details

To guarantee that pattern matching for monadic computations will behave analogously to the standard pattern matching, the implementation of the *choose* operation needs to follow some basic principles. Due to the complexity of the operation, we formulate the rules only informally:

- **Generalized join.** When we apply the operation to a single element list containing computation that yields `Success`, it should behave as monadic *join*.
- **First match.** When there are multiple clauses that are matching on the same monadic value, then the *choose* operation should always choose the first succeeding value in the list. This for example means that the following simple equation should hold:

  choose [map (\x -> Success(_ -> $expr_1$) m);
  map (\x -> Success(_ -> $expr_2$) m)]
  ≡ map (\x -> Success(_ -> $expr_1$) m)

- **One clause.** In the same scenario (multiple clauses matching on the same monadic value), the *choose* operation should not execute multiple clauses. For an impure language, this means that only side-effects of one clause will be executed.

The last two rules are closely related, but we formulated them separately, which makes the discussion in section 8.2 more apparent. The next section discusses the first rule in detail.

$$\begin{aligned} \text{join (return } m) &\equiv m \\ \text{join (map return } m) &\equiv m \\ \text{join (map (}\backslash x \rightarrow \text{join } x) \; m) &\equiv \text{join (join } m) \end{aligned}$$

$$\begin{aligned} \text{choose [return } m \text{]} &\equiv m \\ \text{choose [map return } m \text{]} &\equiv m \\ \text{choose [map (}\backslash x \rightarrow \text{choose [}x\text{]) } m \text{]} &\equiv \text{choose [choose [} m \text{]]} \end{aligned}$$

Figure 14. Monad laws formulated using *join* and *choose*

5.2 Generalization of join

We already stated that the *choose* operation should be a generalization of *join* and that it should behave as *join* in the basic case. The following code shows how to implement *join* if we already have *choose*:

```
let join m =
  choose [ map (fun c -> Success(fun () -> c)) m ]

val choose : list<M<MaybeDelayed<M<'a>>>> -> M<'a>
val join   : M<M<'a>> -> M<'a>
```

As a result, we can define a monad that supports pattern matching in terms of *choose*, *map*, *return* and *merge* (for non-commutative monads). We no longer need *join*, because it is just a special case of *choose*.

The new set of operations should still follow standard monad laws, so we need to specify what laws should hold for the *choose* operation. We'll start with a version of monad laws for the monad definition that uses *join*, *map* and *return* from [45]. The laws are shown in Figure 14 (top).

To get a new set of laws, we could simply replace *join* with $choose \cdot L \cdot (map\ D)$, where L is a function that creates a singleton list and D is a function that wraps an argument into a delayed `Success` value. This corresponds to the implementation of *join* given above. However, this simple syntactic transformation doesn't convey any useful intuition. We can show a simpler and more intuitive version of laws if we work with a *choose* function, which assumes that all pattern matching succeeds:

```
val choose : list<M<M<'a>>> -> M<'a>
```

Monad laws for *join* don't take `Failure` cases into account, so we don't lose any information. The laws that should hold for this simplified choose operation are shown in Figure 14 (bottom).

6. Reasoning about monadic matching

We can use the laws for *merge* and *choose* operations and the translation described in the Section 4.2 to show many useful facts about the `match!` construct. In this section, we look at several that are important for building the intuition about `match!` construct and at some showing that `match!` shares important properties with the usual ML pattern matching.

Generalized binding. When using pattern matching on a single monadic value with a single clause that consists of a variable binding pattern, the `match!` construct behaves like bind:

match! *m* **with** | !*var -> expr*	≡	**let!** *var* = *m* *expr*

From the translation, we can see that the *m* value is passed as an argument to *map*, which produces `Success` (variable pattern never fails). The result is wrapped into a singleton list. In that case, *choose* behaves as *join* (Section 6.2) so the left-hand side becomes a definition of *bind* in terms of *join* and *map*.

Reordering. The result of `match!` will be the same if we reorder its arguments and patterns. The following expression will give the same result for any permutation *p* of *n* elements:

match! m_1, … , m_n **with** | $cpat_{1,1}$, … , $cpat_{1,n}$ -> $cexpr_1$ | … | $cpat_{k,1}$, … , $cpat_{k,n}$ -> $cexpr_k$	≡	**match!** $m_{p(1)}$, … , $m_{p(n)}$ **with** | $cpat_{1,p(1)}$, … , $cpat_{1,p(n)}$ -> $cexpr_1$ | … | $cpat_{k,p(1)}$, … , $cpat_{k,p(n)}$ -> $cexpr_k$

By analyzing the translation, we can see that the permutation only changes the order of *merge* operation applications. However, associativity (C2) and commutativity (C3) laws of *merge* allow us to reorder arguments in any way, so this change does not change the meaning of the expression.

Matching on returns. Next we look at a special case when the arguments of `match!` are constructed using *return*. We can translate the code into a usual `match` (inside a computation expression):

match! m { **return** e_1 }, m { **return** e_2 } **with** | !var_1, !var_2 -> *cexpr*	≡	**match** e_1, e_2 **with** | var_1, var_2 -> *cexpr*

The two computation expressions are passed as arguments to the *merge* operation. Using the (C1) law, we get a single computation expression that returns a tuple. In case with single clause and patterns that can't fail, *choose* behaves as *join*, so we can rewrite the expression as follows:

join (map (\(var_1, var_2) → *cexpr*) (return (e_1, e_2)))

Using the monad laws, we can further simplify this expression. As a result, we get *cexpr* with e_1 and e_2 substituted for var_1 and var_2, respectively, which is equivalent to the `match` expression on the right-hand side.

Unused arguments. Another fact is that we can add an additional argument to `match!` and add an ignore pattern to a pattern list of each clause without changing the meaning of the expression:

match! m_1, …, m_n **with** | $c_{1,1}$, …, $c_{1,n}$ -> $cexpr_1$ | … | $c_{k,1}$, …, $c_{k,n}$ -> $cexpr_k$	≡	**match!** m_1, … , m_n, *m* **with** | $c_{1,1}$, … , $c_{1,n}$, _ -> $cexpr_1$ | … | $c_{k,1}$, … , $c_{k,n}$, _ -> $cexpr_k$

In this rule, $c_{i,j}$ represents any computation pattern. On the right-hand side, we added a new monadic value *m* and a computation pattern "_" to each clause. We can easily see that this rule holds simply from the translation. There is no *binding pattern* for the argument *m*, so it will not appear anywhere in the translated code.

Identity. We can also write a `match!` expression that transforms any monadic value into an equivalent value. Thanks to the previous rule, this is true even if we add additional arguments and *ignore patterns* to appropriate locations.

match! *m* **with** !*v* -> **return** *v* ≡ *m*

Since the pattern matching never fails in this expression, we can rewrite the code using a variant of choose from Section 6.2 which assumes that all pattern matching succeeds. Then we get *choose* [*map return m*]. The second law in Figure 14 states that this is equivalent to *m*.

First match. In the usual ML pattern matching, the compiler can identify clauses that will never be matched. This is also an important aspect of intuition about the `match` construct. For `match!` the following two expressions are equivalent:

match! *m* **with**
| !var_1 -> <*cexpr*>$_1$
| !var_2 -> <*cexpr*>$_2$

≡

match! *m* **with**
| !var_1 -> <*cexpr*>$_1$

This guarantees that the intuition about unreachable clauses is, to some extent, also valid for the `match!` construct. The equivalence is a consequence of the second requirement for the *choose* operation from Section 6.1 and of the translation.

We've looked at several facts that hold about the `match!` construct. As we can see, many of the facts directly correspond to the usual intuition about standard ML-style pattern matching, which is the key goal of our design.

7. Design alternatives and future work

In this section we look at various design alternatives of monadic pattern matching that may be relevant for other types of computations or other programming languages. We'll also discuss some problems that we don't tackle in our current design and may be interesting in the future.

7.1 Representing failure inside monad

Our design represents pattern matching failure explicitly outside of the monadic type using the `MaybeDelayed<'a>` type. This type also delays the computation, which allows us to evaluate pattern matching of the clause without evaluating side-effects of the body. An alternative option would be to represent the failure inside the monad using an additional *zero* operation provided by some monads (also called *fail* in Haskell). For example, a reasonable representation of failure in the List monad is an empty list `[]` and the `Maybe` monad uses the `None` case (named `Nothing` in Haskell).

One problem with this approach is that we would be able to use `match!` only with monads that can represent failure. We can't use any default implementation (such as throwing an exception) in this case as the failure is a legitimate result. It simply informs the *choose* operation that it needs to select another clause. How-

ever, a more important problem is that we need to decide how exactly to represent the failure. A clause takes an input of type `M<'a>` into a result of type `M<M<'b>>`, so we could represent the failure either in the inner or in the outer monadic value. However, our goal is to keep the semantics close to the ML pattern matching so none of the two options is works well for us.

Inner value. When using the inner value, the body of a clause would evaluate to a failure value inside the monad. In this case, we could still use the *map* operation in the translation (Section 3.2). The following example shows how the `match!` Construct would behave if we used this approach:

```
maybe { match! Some(1) with
        | !1 -> printf "one";  return! None
        | !_ -> return 42 }
```

We're using `None` to represent the pattern matching failure. In case of success, we return the computation representing the body of the clause. The translation looks like this:

```
choose [
  map (function 1 -> printf "one"; None })
              | _ -> None) (Some(1));
  map (function _ -> Some(42)) (Some(1)) ]
```

When executing this example, the `choose` function starts evaluating the clauses to find the first one that succeeds. The pattern of the first one matches, so it doesn't return `None` immediately. It returns some computation, but the computation may still return `None`. In our case it does that after running some side-effect. As a result, the choose function needs to continue searching and it finds the second clause which returns 42 as the result.

As we can see, this approach doesn't fit well with the usual ML pattern matching, which selects the first succeeding clause. In this example, a clause starts evaluating and then fails at some later time, which resumes the pattern matching.

Outer monad. In this case, we'd modify the translation to use bind instead of map in Figure 12 (2). This, however, changes the structure of the outer monad, which makes it impossible to implement the *choose* operation for some monads. The reason is that the *choose* operation may rely on the structure in some way (and in particular, the structure should be the same for clauses that pattern match on the same monadic value).

However, when using this alternative, we would use *bind* instead of map in the translation (see rule (2) in Figure 12) and we would use additional return instead of `Success` and monadic failure in place of `Failure`. This would break the behavior of `List` that uses the zip function as merge operation:

```
> list { match! [ Left 1; Right 2; Left 3 ] with
         | !Left n -> return n
         | !_ -> return 0 };;
val it : list<int> = [ 1; 3; 0 ]
```

The computation replaces all values tagged with the `Right` case with a default value 0 and the expected result would be [1; 0; 3], however, due to the encoding of

failure in the outer monad, we get an unexpected result. The reason is that the first clause produces a list [1; 3] which has a different structure (length) than the input. The second clause produces a list [0; 0; 0] and the choose operation combines them into the unexpected result [1; 3; 0].

We believe that our approach is the only viable choice for a language from the ML-family. However, for a pure language without side-effects (most importantly Haskell), the representation of failure in the inner value may be very well reasonable. The representation using the outer value would restrict the number of monads that can provide a reasonable implementation of the `match!` feature.

7.2 Monadic Active Patterns

Active patterns [39] were introduced as a mechanism for creating abstractions over algebraic data types. This has been a well-understood problem since [40]. Our generalization is orthogonal these proposals, because it focuses on the pattern matching process as a whole rather than on the individual patterns. We can use active patterns in the usual way as part of our binding patterns:

```
match! mv with | !Polar(m, p) -> ...
```

This code matches on the monadic value `mv` representing a complex number and views it in a polar form using total pattern from [39]. However, [39] and [65] also propose a possible generalization for monadic pattern matching. It uses active patterns that take a value and return a monadic type `M<'a>`. The following example uses active pattern named `Id` which simply returns its argument:

```
matchm<List> [0; 1], [2; 3] with
| Id 0, Id y -> y
| Id x, _ -> x
```

The proposed desugaring uses the Haskell's `MonadPuls` type class. This is appealing, because `MonadPlus` is well-known and widely used type. The previous example would be translated as follows:

```
mplus
 (m { let! x = Id [0; 1] in let! y = Id [2; 3] in
      if x = 0 then return y else return! mzero })
 (m { let! x = Id [0; 1] in return x })
```

If we executed the example, the result would be [2; 3; 0; 1]. This is quite different to the behavior of our examples presented earlier. The reason is that encoding using `MonadPlus` executes all clauses for which the pattern matching succeeds. This may be the desired behavior for Haskell, but not if we want to follow the usual intuition about the ML pattern matching. In particular, this example doesn't follow the *First match* rule and would also break the *Reordering* rule for monads that are not commutative.

Our encoding gives the author of monad more freedom and guarantees the usual ML intuition, provided that the author obeys several simple laws. It is also possible to use pattern matching to express operations that are not monadic (such as zipping of lists), but are complementary.

7.3 Executing Active Patterns once

Our extension can be used together with active patterns. Since active patterns may contain side-effects, it is important to analyze how many times the pattern will be run. Using the rules from Figure 12, each clause is translated into a separate monadic value containing independent `match` expression. As a result an active pattern will be evaluated separately for every clause. As stated in [46], a more reasonable semantics is to execute each active pattern at most once. We can achieve that by lifting all patterns that match on one argument of `match!` into single pattern matching. We can evaluate all patterns once at the beginning of `match!` by mapping each argument of `match!` into results of all patterns using the monadic map function:

let *f1'* = map (**function**
| p_1 & … & p_n -> Some(vs_1), …, Some(vs_n)
| p_1 & … & p_{n-1} -> Some(vs_1), …, Some(vs_{n-1}), None
| p_1 & … & p_{n-2} & p_n -> Some(vs_1), …, None, Some(vs_n)
| …) *f1*

In this pre-processing rule, p_1 … p_n are patterns that the `f1` value is matched against; vs_1 … vs_n are tuples (possibly of different lengths) of free variables of the corresponding patterns. We need to create a single clause for each subset of the set of all patterns, because we need to identify all patterns that the value matches, so the number of clauses is n^2. The need to identify free variables makes the compilation more difficult.

On the other hand, this compilation may execute patterns that are not needed later (if the pattern occurs in a clause, but an earlier clause succeeds). This appears to be unavoidable, because a monadic value may utilize parallelism.

7.4 Compile-time pattern checking

Our compiler prototype doesn't implement compile-time analysis of pattern matching redundancy and incompleteness. However, to discuss this problem, we need to distinguish between two kinds of monads. Some monads will immediately or eventually have an actual value for each monadic value used as an argument (for example `Future` or `RandomMonad`). Other monads may never produce an actual value from certain monadic values (for example `Maybe` or `List`). To provide maximal guarantees, we'd need to handle these two options slightly differently:

- **Values eventually available.** In this case we need to check whether the patterns enclosed inside our *binding patterns* are exhaustive. An *ignore pattern* handles all cases, so it can be treated as a usual *underscore pattern*.
- **Value possibly missing.** When a value may be missing, we need to cover a case when only a single value becomes available, so we need a clause with only a single binding pattern for every monadic argument.

The possibility of adding compile-time checking is in more details discussed in section 2 of Chapter VII. It also clarifies why we cannot handle a case when no value is available (in the second type of monad).

8. Chapter summary and related work

We demonstrated how to enrich monadic computations (especially in the F# language) with the support for pattern matching on monadic values. In this section, we discuss related types of computations, related work that generalizes pattern matching and work that inspired the applications of our extension.

8.1 Computation types

Apart from monads [29], there are several other types of computation models. Applicative functors [42] provide an abstraction that is more common than monads, but less powerful. As far as we know, applicative functors haven't yet been used in the area of reactive and parallel programming. Co-monads (a categorical dual of monads) [52] have shown useful for data-flow programming [53].

Arrows [47, 48] are used mainly in functional reactive programming research, which is based on working with time-varying values and discrete events. Our examples of reactive programming mostly focused on discrete events. Arrow computations can be written using the arrow notation [48] and we believe it may be interesting to consider whether generalized pattern matching could be provided in the notation as well.

8.2 Pattern matching

The work on pattern matching mainly focused on providing better abstraction when pattern matching on normal values [40, 46]. However, some authors proposed a possible extension, which is to generalize the return type of a pattern from `Maybe` to any instance of Haskell's `MonadPlus` type class [39, 65] as discussed in section 7.2. Although this may be a possible approach, we choose to define a new set of primitives. This gives us more freedom when defining monadic pattern matching and it also allows us to preserve the intuition about pattern matching in ML-family of languages.

Extensible pattern matching in Scala [50] is more powerful, because patterns are represented as objects that can be combined using user-defined operators. This way Scala also provides a very elegant syntax. In F#, we could achieve similar effect by allowing definitions of active patterns as members of an object type. This would be a desirable extension that could be nicely integrated with the work presented in this chapter.

8.3 Applications

We have demonstrated that our monadic pattern matching can be used for encoding a wide range of programming models. Join patterns can be also implemented using extensible pattern matching in Scala [36]. A notable difference is that our implementation is based on asynchronous workflows [4], which is a monadic computation. This allows us to avoid blocking threads when waiting, which is a very important property on platforms where creating threads is expensive (such as .NET and JVM). Our encoding supports nested patterns as described in [51]. However, as noted in [33], it is difficult to efficiently implement pattern matching on general patterns in joins (especially in the presence of guards). Our prototype implementation assumes that guards are not used, although the compiler cannot

forbid their use. Allowing the author of the monad to disallow the use of guards seems like a useful extension.

Other applications that we demonstrated in this paper is the reactive programming library, which is based on discrete events (as introduced in [2]) and uses imperative programming encoded using monads (similarly to [3]). We also presented parallel programming model based on futures [37]. Pattern matching in this model is very similar to the `pcase` (parallel case) introduced in Manticore [38].

8.4 Conclusion

The key claim of this paper is that a wide range of reactive and concurrent programming models can be encoded using a simple reusable language extension, without the need to design a specialized language for each programming model. We presented a language feature that extends F# computation expressions with monadic pattern matching and we sketched numerous applications ranging from lists processing to concurrent programming.

Our language extension is based on pattern matching construct known from many functional programming languages. We integrated it in the F# language, which is based on ML, so we made a special effort to preserve the user's existing intuition about pattern matching. By requiring several simple laws about basic combinators, we can guarantee numerous results that are helpful for reasoning about our monadic pattern matching. Aside from practical applications, our work is also shows an interesting relation between monadic pattern matching and commutative monads. In particular, our construct can be used for binding on multiple monadic values in parallel using a syntax that is less sequential than the one used by standard monads.

Chapter VI

Reactive event-driven computations

In this chapter, we introduce our imperative programming model that complements the declarative programming model presented in Chapter II. Thanks to the improved implementation of F# event combinators (described in Chapter IV) and to the generalized pattern matching extension (Chapter V), we get a universal and very powerful way for programming reactive applications.

Programs encoded using our programming model are structured as a set of concurrently executing processes that may wait for events and perform some reaction in response to an event. Each computation constructs an event that can be triggered only from within the computation. The program is then composed by connecting these events. Although the processes can be viewed as concurrent, the actual implementation is single-threaded, which makes it possible to reason about the programming model. We argue that simple user interface interactions do not need actual parallelism, because CPU intensive computations should not be performed on the user interface thread at all.

More specifically, this chapter presents the reactive programming model using a series of examples and makes several other contributions to the theory and practice of reactive programming:

- We demonstrate our programming model using a series of examples that introduce individual features (Section 1) and use it to develop a simple reactive game to demonstrate the practical benefits of the model (Section 2). We show that the programming model makes it easy to encode state machines and that it provides an elegant abstraction for structuring applications.
- We develop a notion of semi-discrete time, which is particularly suitable for modeling reactive event-driven applications that are not based on the synchronous programming model. We use it to present a formal semantics of the programming model (Section 3) and we use it to formally show several useful properties about the programming model (Section 4).
- We analyze the event builder computation from an abstract point of view. As noted earlier, it doesn't form a monad, because of its imperative natures. This type of computations is, however, useful in practice. We present a computation for working with mutable sequences of values (Section 5.1) and describe the type of computations abstractly using algebraic laws (Section 5.3).

1. Reactive library by example

In this section, we will gradually introduce our reactive programming library using a series of examples that demonstrate the key features. Our library works with an improved representation of events called `EndigEvent<'a>`, which is very similar to `IEvent<'a>` from F# core libraries with the addition that it can notify the listeners that it finished producing values. We'll discuss the type in section 1.5.

1.1 Limiting mouse clicks

In our first example, we will take an existing event `win.MouseDown` of the type `EndingEvent<MouseEventArgs>`. The event is triggered each time the user clicks on the window. We'll create a derived event of the same type, which will be also triggered on click, but at most once a second:

```
1: let limitedClicks =
2:   let rec loop () = event {
3:     let! arg = win.MouseDown
4:     yield arg
5:     do! Event.after 1000
6:     yield! loop () }
7:   loop ()
```

The example uses the `event` computation builder and presents four constructs of computation expressions. On the line 3, we use the *bind* operation. It waits for the first occurrence of the `MouseDown` event and then resumes the computation, assigning the argument carried by the event to the `arg` value. The operation ignores any subsequent occurrences, so the rest of the computation will be executed (at most) once.

The line 4 uses the *yield* operation. It triggers the returned event with the value of `arg` as the carried argument. Next, we want to wait one second before handling any further events. This is done on the line 5, by calling the `after` function. It returns an event that will occur after the specified time. The event carries `unit` value as the argument, so we can use the `do!` syntax, which is similar to `let!`, but ignores the result. The line 6 recursively invokes the `loop` function, which starts waiting for the next occurrence of the `MouseDown` event. Finally, we invoke the `loop` function to create a single value of the constructed event with limited click frequency (line 7).

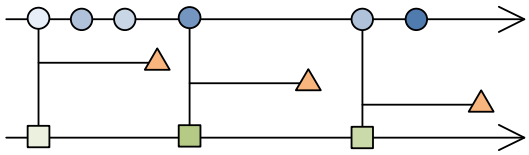


Figure 15. Circles represent mouse click events; triangles represent events triggered by the `after` function; squares depict the occurrences of the resulting event.

The behavior of the event is depicted in the

Figure 15. The upper horizontal line with circles shows the occurrences of the `MouseDown` event and the lower line with rectangles shows the constructed event. The lines in the middle correspond to the events created by the `Event.after`

function. This simple example demonstrates two important aspects of our reactive library:

- **Missing events.** The occurrences of events are not cached in any way and so it is possible to miss a value if the event occurs while all computations (constructed using the `event` builder) are waiting for another event.
- **Active events.** To construct an active event, we just need to create a value of the event. The computation is executed automatically when the application starts. Our example demonstrates a common pattern where we create a processing loop as a recursive function and then run the function to create an active event value.

1.2 Waiting for multiple events

Our second example demonstrates the `match!` keyword, which allows us to wait for a combination of events. For example, let's say that we want to implement an application for drawing shapes. The user can create a rectangle by pressing the button, moving the mouse and then releasing the button again.

We can safely assume that no button is pressed when the application starts and that `MouseDown` and `MouseUp` occur in an interleaving order. Then we can implement drawing using just a single `match!` construct as follows:

```
1: let rectangles =
2:   let rec loop () = event {
3:     match! f.MouseDown, f.MouseUp with
4:     | !down, !up ->
5:       yield (down.X, down.Y), (up.X, up.Y)
6:       yield! loop () }
7:   loop ()
```

The `match!` construct on line 3 takes two input events. The pattern matching has only one clause (line 4), which consists of two *binding patterns* (the "!" symbol followed by a standard F# pattern). This specifies that we need to wait for an occurrence of both of the events before executing the body.

Similarly to the *bind* operation, pattern matching waits for the first suitable combination of event occurrences and then executes the first matching clause (at most) once. When this happens, we yield the coordinates of the newly created rectangle (line 5) and recursively call the `loop` function to wait for the next rectangle (line 6). This example demonstrates:

- **Joining events.** In reactive applications, we often need to wait for a combination of events. The generalized pattern matching construct isn't fully universal, but allows us to express many frequently used patterns. We will discuss the semantics of `match!` in more details in section 3.5.

1.3 Creating stateful events

The event created in the previous section is triggered each time the user enters a new rectangle. However, in reality we will need to collect all the created shapes, so that we can draw them when the window is invalidated. When creating events, we can store the state as a parameter of the `loop` function:

```
1: let rectangleList =
2:   let rec loop rects = event {
3:     let! rc = rectangles
4:     yield rc::rects
5:     yield! loop (rc::rects) }
6:   loop []
```

The parameter `rects` stores the list of rectangles drawn so far. When a new rectangle is drawn (line 3), we first trigger the created event with the new list as an argument (line 4), and then we continue looping with the new state as the parameter. Note that when starting the loop (line 6), we need can provide an initial value of the state. This example again demonstrates a couple of interesting aspects of our library:

- **Composability.** In this example, we're using an event value created earlier (namely `rectangles`) to build a more complicated event. Although our library has in many ways imperative nature, it is highly composable. We discuss laws that are useful for reasoning about the composition in section 5.
- **Guarantees.** Our library provides certain guarantees (section 4) that specify when an event value may be missed. For example in the code above, we will never miss a value produced by the `rectangles` event.
- **Stateful events.** As shown by the example, we can write events that keep an internal state. The state exists exactly once, which means that all computations that will be waiting for an occurrence of the `rectangleList` event will always receive the same value.

1.4 Transitioning between modes

So far, our sample application doesn't give any visual feedback when drawing the rectangle. In this section, we'll create an event that fires repeatedly when drawing the rectangle. The event has two different modes of operation. In the first one, it waits for the `MouseDown` event. In the second state it fires each time the mouse pointer moves.

```
1: let selection =
2:   let rec drawing down = event {
3:     match! f.MouseMove, f.MouseUp with
4:     | !move, _ ->
5:         yield down, (move.X, move.Y)
6:         yield! drawing down
7:     | _, !up ->
8:         yield! waiting() }
9:
A:    and waiting () = event {
B:     let! down = f.MouseDown
C:     yield! drawing (down.X, down.Y) }
D:   waiting()
```

The two different modes of the event are implemented as two mutually recursive functions. The function `drawing` (line 2) corresponds to the mode in which the computation yields the coordinates each time the mouse moves (line 5) and the function `waiting` (line A) represent a mode in which the computation awaits the `MouseDown` event (line B). The transition between different modes is implemented as a recursive call (using the `yield!` primitive) in the tail-call positions of the two

functions (lines 6, 8 and C). Note that we need to specify the initial mode when starting the computation (line D).

In the `drawing` mode, we use `match!` to encode another frequently used pattern, different to the one in section 2.3. We need to wait either for `MouseMove` or `MouseUp` event, whichever occurs first. This is done using two clauses. The first one (line 4) uses binding pattern to obtain a value from `MouseMove` and ignores the other event, while the second clause (line 7) obtains a value from `MouseUp`. Let's briefly summarize the most interesting aspects of our last example:

- **Choosing events.** This example demonstrates that `match!` can be used not only for joining multiple events, but also for choosing between them. We can freely combine these two uses, which makes `match!` very useful.
- **State machines.** In the code above, we wrote a computation with two modes using a technique that can be, in general, used for encoding arbitrary finite state machines. We can use mutually recursive functions just like when encoding state machines in ordinary functional programs.

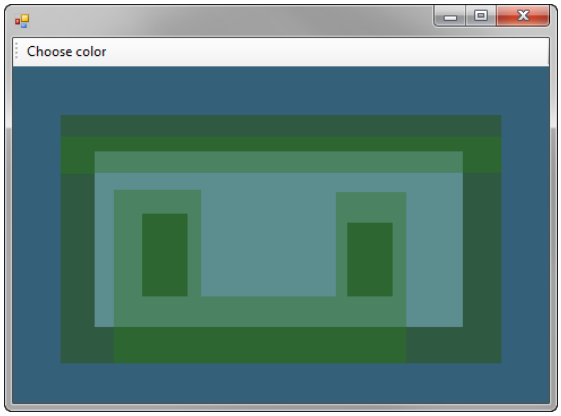


Figure 16. Drawing rectangles in action

The last three code snippets form a significant part of a working demo application for drawing rectangles. The full source code available on the attached CD contains a simple addition that allows the user to select the rectangle color, which is demonstrated in **Figure 16**[9].

1.5 Ending event

When writing code using the event builder computation, we need to equip the computation builder with the `Yield` member to trigger an event, but also with the `Combine` member that first runs one computation and when it completes, starts the second computation. In order to implement this member, we need to know when the event produced the last value (indicating that we can start running the second one). For this reason, the `IEvent<'a>` type isn't sufficient.

As already mentioned, our computation builder constructs values of a type `EndingEvent<'a>` which extends event with the ability to notify the listeners when the event has completed. Additionally, we also add notification about an exception:

[9] We are grateful to Josef Albers for making it possible to demonstrate the application using an art reproduction without adding support for other shapes.

```
1: type EventStatus<'a> =
2:   | Completed
3:   | Value of 'a
4:   | Exception of exn
5:
6: type EndingEvent<'a> =
7:   | Ending of IEvent<EventStatus<'a>>
```

The declaration uses the standard F# event type as a basis. The `EndingEvent<'a>` type is a simple wrapper that encapsulates a standard event (line 7) that produces values of the `EventStatus<'a>` type. This type represents various messages that the event may produce. It includes a case when the event produces a standard value (line 3) and two special cases representing the end of the event computation (line 2) and an exception (line 4).

A valid event computation should follow a simple contract. It should produce zero or more notification carrying the `Value` status and then it may eventually produce either `Completed` notification (when it finished successfully) or the `Exception` notification (when some error occurs). After that, no further notification should be produced.

2. Case Study: Simple reactive game

In this section, we will demonstrate our reactive library using a more complex example. This will demonstrate several interesting aspects, such as modelling of data flow in reactive programs (Section 2.1) and the ability to hierarchically compose finite state machines.

Figure 17. Simple reactive game in the playing mode

We will implement a game where the user has to click on a moving smiley face as many times as possible within 20 seconds. The smiley moves repeatedly after some short time or immediately after it is clicked. In addition, when a smiley is clicked, it changes color for a short period of time. As demonstrated in **Figure 17**, the game also shows the remaining time and the current score (number of clicks). After 20 seconds, the game asks whether the user wants to play again.

2.1 Gameplay data flow

We start by implementing parts of the game that are active while the user is playing. This phase can be decomposed into three individual components. Each component can be encoded as an event. The components can communicate by

waiting on values produced by other components (events). This is possible thanks to the composability discussed in section 2.4. As we'll see later, the gameplay will be started repeatedly, so we need to make sure that all the involved components end running when the remaining time reaches zero. This can be done using an aspect that we haven't discussed yet.

In section 3, we model events as a sequence of time/value pairs associated with a time range when the event runs. This means that an event can end and stop producing values. All examples in the section 1 were recursively implemented infinite events. However it is possible to remove the recursive call and create event that ends. In the implementation of the library, we can also wait for a special notification that is triggered when an event ends.

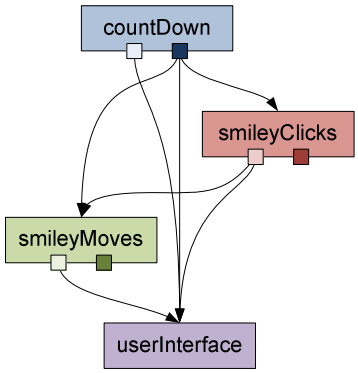


Figure 18. Data flow of game components; the output drawn as a box on the left side represents a produced value; the right output is triggered at most once when the component completes.

Figure 18 shows individual events of the Gameplay. It demonstrates how values flow between the events. The arrows in the diagram have the following meaning:

- An arrow from the left output of an event A to an event B corresponds to the fact that the event B waits for occurrences of the event A.
- An arrow from the right output means that the event B waits for a notification triggered when the event A ends

Now that we have an overall picture of the Gameplay architecture, we can discuss individual components in more details:

- **CountDown** event produces a value storing the number of remaining seconds (an integer) each time the number changes. The event ends after 20 seconds.
- **SmileyClicks** reacts to the `MouseDown` event from the user interface and fires each time the user clicks on the smiley. The carried value is the number of clicks so far. This event keeps running until the `countDown` event ends.
- **SmileyMoves** yields a location of the smiley and its state (normal or red face). A new value is produced when the user clicks on the smiley or when the smiley changes state or moves after a specified time. The runs until it receives notification that `countDown` event ended.

From these three events, only the third one implements some complicated behavior that we'll need to express using a state machine. The first two events are very simple.

2.2 Counting time and clicks

We first look at the `countDown` and the `smileyClicks` events. As you can see from the **Figure 18** the `countDown` event doesn't have any input links and so it is the best one to start with. The `smileyClicks` event waits until the `countDown` event ends, so we'll implement it as the second one:

```
1: let countDown =
2:   let start = DateTime.Now
3:   let rec loop(n) = event {
4:     yield n
5:     if n > 0 then
6:       do! Event.after 1000
7:       yield! loop(n - 1) }
8:  loop(20)
```

The code follows the pattern discussed in section 2. We create a recursive loop that keeps the number of remaining seconds as the parameter. Inside the loop we trigger the event with the current number (line 4), wait one second (line 6) and then loop recursively (line 7). When the number of seconds reaches zero, the event ends (the implicitly added `else` clause contains an empty event that immediately ends). The `smileyClicks` event will be slightly more complicated as it needs to handle two cases. When the `countDown` event ends, it also needs to end. When the user clicks on the smiley it produces a new score value:

```
1: let smileyClicks =
2:   let rec loop(n) = event {
3:     match! smiley.MouseDown,
4:            countDown.Completed with
5:     | !md, _ when hitTest md ->
6:       yield n + 1
7:       yield! loop(n + 1)
8:     | _, !_ -> () }
9:   loop(0)
```

The event is again implemented as a `loop` function. We start the function with an initial score set to zero (line 9). The event uses `match!` to wait for the first of two events. When the user clicks on the smiley control, we need to check whether the mouse cursor was actually inside the smiley circle. This can be done in the `when` clause (line 5) using the `hitTest` function (not shown in the code). Occurrences of the `MouseDown` event that don't match this condition are ignored. When the location of the cursor is correct, we produce a new score value (line 6) and then continue looping (line 7).

The second event that we're waiting for is the notification that the `countDown` event ended. This notification is exposed as the `Completed` member. The member is a standard event and fires exactly once when the source event ends. The branch that handles this case contains a unit value (line 8), which means that it doesn't do anything and so the `smileyClicks` event also ends.

The above code is correct only if we can guarantee that the `Completed` event will not occur unnoticed, for example when handling a click on the smiley. Our library guarantees that if code doesn't perform any waiting (using `match!` or `let!`),

no values will be missed while running, which makes the above implementation correct. This guarantee is in more details discussed in section 4.

2.3 Calculating smiley state and location

Next, we implement the event that represents the state of the smiley face. The event will trigger when the smiley moves or when it changes color. These changes can be caused by either a click on the smiley or by a timeout. The control flow is slightly more complicated, so we will implement it as a state machine. The diagram in **Figure 19** shows the state machine.

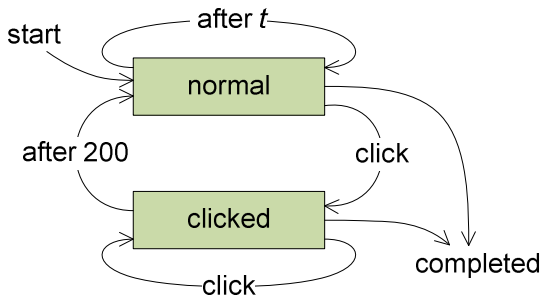


Figure 19. State machine representing the `smileyMoves` event that controls location and state of the smiley

The event starts in the *normal* state. There are three different transitions from this state. After some time, the smiley moves automatically. This triggers some action and changes the state back to *normal*. If the user clicks on the smiley, it moves immediately and transitions to the *clicked* state, which changes the color of the smiley to red. In this state, the user can either click on the smiley again. After 200 milliseconds, the color of the smiley changes back to normal (without moving the smiley again). In both of the states, we also handle the case when `countDown` event ends and in response we terminate the `smileyMoves` event.

We'll implement the state machine using mutually recursive functions. As we can see in the previous diagram, the two states share similar structure and so the two functions also look similar (we could refactor the code using higher-order functions, but we show the straightforward solution to keep the code simple and readable):

```
A: let smileyMoves =
B:   // Moves smiley to 'pos' and changes color to
C:   // normal, then waits for the specified time
D:   let rec normal(pos, time) = event {
E:     yield normalSmileyImage, pos
F:     match! countDown.Completed, smileyClicks,
G:            Event.after time with
H:     | !_, _, _ -> ()
I:     | _, _, !_ -> yield! normal(newPoint(), 800)
J:     | _, !_, _ -> yield! clicked() }
K:
L:   // Moves smiley to a random location and
M:   // changes color to red, then waits 200 ms
N:   and clicked() = event {
O:     let pos = newPoint()
P:     yield redSmileyImage, pos
```

```
Q:      match! countDown.Completed, smileyClicks,
R:             Event.after 200 with
S:      | !_, _, _ -> ()
T:      | _, _, !_ -> yield! normal(pos, 600)
U:      | _, !_, _ -> yield! clicked() }
V:
W:    // Start with a new random location
X:    normal(newPoint(), 800)
```

The function representing the *normal* state takes a position and a remaining time until the next movement as arguments (line D). This allows us to call the function recursively from the *normal* state where we want to move smiley to a new location and wait 800ms before moving again (line I), as well as from the *clicked* state where we want to keep the same location and wait only the remaining 600ms after already waiting for 200ms (line T).

In both of the functions, we use `match!` to wait for the first of three events (lines F, Q). This is perhaps the most common pattern when programming simple reactive user interfaces. One notable aspect of the implementation is that it repeatedly creates a new event using the `Event.after` function (lines G, R). This differs from the usual use when we wait for existing events. However the `Event.after` function serves more as a primitive of the library and it will by, indeed, defined as one of the primitives in the formal semantics (Section 5).

Also note that we use the "!_" pattern in all the clauses, which means that we're waiting for the event, but we're discarding the carried value. For example, the `smileyClicks` event carries the current score, but we don't need this value in the `smileyMoves` event. We only need to know that the user clicked on the smiley (and so the score has changed).

The event produces tuples of type `Bitmap * Point` (lines E, P). Inside `normal`, the bitmap is `normalSmileyImage` and the position is the one given as the parameter. Inside the `clicked` function, we always yield `redSmileyImage` together with a newly generated random location (the function `newPoint` is not shown in the listing).

2.4 Hierarchical structure of the game

In the previous sections, we implemented the events that run during the Gameplay. In this section, we'll look how to embed them in the rest of the application which provides user interface for starting the game and displaying the score when the game ends. The overall structure of the application is very simple and is displayed as a state machine in **Figure 20**.

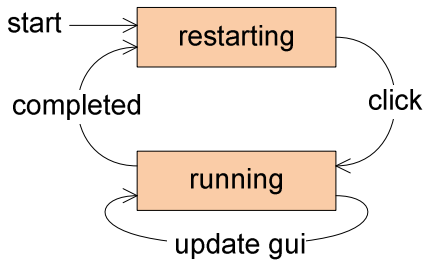


Figure 20. The game loop visualized as a state machine

The game runs a very simple loop. When it starts, it enters the *restarting* state in which it displays some information and waits unit the user clicks button to start playing. Then it transitions to the *running* state in which it repeatedly updates the user interface based on the instructions from the game. When the game ends, the total score is displayed and we transition to the *restarting* state where the user can start a new game. As already mentioned, we construct a new instance of the events `countDown`, `smileyClicks` and `smileyMoves` each time the user starts a new game. This solution is correct because we also make sure that all of these events stop running at the end of the game. To start a new instance of the events, we wrap the code that we wrote so far in a function that returns the constructed event values:

```
let startGame() =
  // (definitions from sections 4.2 and 4.3)
  smileyMoves, smileyClicks, countDown
```

We will invoke this function when transitioning from the *restarting* state to the *running* state to create a new instance of the Gameplay. Once we create the events, we will keep looping until the `countDown` event ends.

To avoid the need to pass all there events as parameters while looping, we declare a nested function `loop` which implements the *running* state. The function remembers the current score, which is needed when the game ends:

```
A: let userInterface =
B:   let rec restarting() = event {
C:     displayGameInfo(true)
D:     let! _ = start.Click
E:     displayGameInfo(false)
F:     yield! running(startGame()) }
G:   and running(e) =
H:     let smileyMoves, smileyClicks, countDown = e
I:     let rec loop(score) = event {
J:       match! smileyMoves, smileyClicks,
K:              countDown, countDown.Completed with
L:       | !(pos, img), _, _, _ ->
M:         smiley.Image <- img
N:         smiley.Location <- pos
O:         yield! loop(score)
P:       | _, !score, _, _ ->
Q:         score.Text <- sprintf "Score: %d" score
R:         yield! loop(score)
S:       | _, _, !time, _ ->
T:         time.Text <- sprintf "Time %d" time
U:         yield! loop(score)
V:       | _, _, _, !_ ->
W:         displayScore(score)
X:         yield! restarting() }
Y:     loop(0)
Z:   restarting()
```

The restarting function is simple. It displays the `start` button and other controls that show the game info (line C) and waits for a click on the button (line D). To start a new game, we call `startGame` to create a triple of events that is passed to the recursively called `running` function.

The `running` function decomposes the tripe into individual events (line H) and starts looping. The first three branches handle events that provide updates for the user interface, namely changing smiley location and picture (line L), updating the current score (line P) and changing the remaining time (line S). The last branch (line V) waits on the `Completed` notification exposed by the `countDown` event. We display the user's score and start waiting for another game.

2.5 Practical lessons learned

Before we move on to discussing the semantics of our reactive programming model, we discuss several properties that were demonstrated by the previous case study and that are important for using the library in practice.

Modularity. The first practical benefit of our library is that we can develop and test individual components (implemented as events) largely independently. The Gameplay part of our application was structured into three largely independent events dealing with time counting, click counting and moving the smiley. In the usual program that suffers from the inversion of control, all these three aspects would be mixed together.

This separation simplifies the development of the components as well as their testing and is also essential for reusability. Note that an event may consume values from another event (for example, `smileyMoves` receives notifications from `smileyClicks` when the user clicks on the smiley). However, we can easily replace the referenced event with a mock implementation that does nothing (when we want to develop a component without other interactions) or simulates behavior of the user (which is invaluable for unit testing).

User interface separation. As the sample game shows, the library allows us to separate the game control logic (implemented in sections 4.2 and 4.3) from the code that manipulates the user interface (in section 4.4). This means that if we wanted to change the presentation layer, we would only have to change the `userInterface` function.

Indeed, we could take this idea even further and hide the entire application behind a façade event that would yield values of some algebraic data type representing instructions to the user interface.

Visualizing the structure. The structure of a program implemented using our library is very easy to visualize graphically. We presented two kinds of diagrams in this paper. The data-flow diagram (**Figure 18**) visualizes how components of the application communicate and the control-flow diagram (a state machine) visualizes the behavior of a component in detail.

The key benefit is that the code we write directly corresponds to the diagrams. This makes it easy to draw diagram for existing code as well as implement a program based on a diagram. As the case study demonstrated, we can also compose the components (and diagrams) hierarchically, which is very useful for applications of larger scale.

3. Formal semantics

In this section, we present formal semantics of our reactive library. The library has many imperative properties (such as waiting for events without caching), so the code that uses it may seem difficult to reason about. Here we clarify what expectations we can have when using the library.

We model an event as a list of time-value pairs. An event (such as `countDown`, `smileyClicks` and `smileyMoves` from the previous section) is constructed using an event-builder computation (consisting of constructs such as `let!` and `yield`). We present a big-step operational semantics of event-builder computations and use it to show when an imperative loop can miss an event occurrence, which was our main concern with respect to semantics in the previous sections.

3.1 Motivating examples

We start by discussing three examples where the expected meaning is intuitively clear and where a naïve semantics (and implementation) may easily go wrong.

Determinism. The first example shows what should happen when we have an event-builder computation with several consecutive `yield` constructs. For example the following computation (which starts immediately when the application runs) yields three multiples of 10:

```
let nums = event {
  yield 10; yield 100; yield 1000; }
```

If another computation waits for the `nums` event, we should guarantee that the numbers will be received in the order in which they were generated. This means that the semantics cannot use the same time value for all three produced values. The following interpretation would be clearly wrong:

*val nums : (time * int) list = [(0, 10); (0, 100); (0, 1000)]*

When waiting for the first value from `nums`, we would pick a value with a minimal time, which can be any of the three produced values. This shows that the computations needs to track the current time and increment it each time it yields a value. A valid interpretation of the above code would be for example a list *[(0, 10); (1, 100); (2, 1000)]*. Here, the time component is incremented by one after each `yield` construct.

Nested yielding. An event-builder computation typically waits for an occurrence of some event and then performs several actions in response. These actions may include waiting for another event or triggering the event (using the `yield` construct). If we write a computation that recursively waits for the same event, then we would like to know what we can do in response if we don't want to miss any event occurrence.

If we wait for another event before waiting again (as for example in section 2.2) then we cannot make any guarantees in general. However, if the reaction consists only of triggering the event, than we would expect the computation to handle all values generated from the source event. Let's demonstrate this using an example (which uses the `nums` value declared above):

```
1: let twice =
2:   let rec loop() = event {
3:     let! n = nums
4:     yield n; yield n * 2
5:     yield! loop() }
6:   loop()
```

The event-builder computation waits for an occurrence of the `nums` event (line 3) and then triggers the `twice` event two times with two different values (line 4). As discussed in the previous example, these two values need to be produced with a different time, so that we can distinguish the first one.

If we simply used one global time for timing occurrences of the `nums` event as well as the `twice` event, the semantics could yield unexpected results. In particular, if processing of the `loop` body (line 4) takes two time units, the meaning of the `twice` event could be following:

*val twice : (time * int) list = [(0, 10); (1, 20); (2, 1000); (3, 2000)]*

As we can see, the list contains only values 10, 20, 1000 and 2000, which shows that the value 100 produced by the `nums` event was missed. The problem is that while producing the two values (on the line 4), we incremented the global time after each `yield`. When waiting for the second value from the `nums` event (after the recursive call), the current time of the event-builder computation was 2 time units. Waiting using the `let!` construct then picks an occurrence with the time greater or equal to 2, which carries 1000 as the value.

One way to solve this problem is to use a different time in the computation that performs the reaction. We call this the *local time*. The timing of the event that we're waiting for (in the example above `nums`) is called an *external time*. The idea is that all operations performed in the *local time* must fit into a single unit of the external time. This guarantees that the entire reaction will only take less than one unit of the external time and so we won't miss any occurrences of the external event. Using this interpretation, the meaning of the `twice` event would be the following:

*val twice : (time * int) list = [(0.0, 10); (0.1, 20);*
(1.0, 100); (1.1, 200) (2.0, 1000); (2.1, 2000)]

As we can see, when running the computation in response to the event `nums` (which occurred at time 0), we increment the local time only by 0.1 for each `yield`. When we start waiting for the next occurrence of `nums`, the local time is 0.2, so we won't miss the value 100, which occurs at the time 1.

Note that the local time isn't a decimal number, as the computation may need to yield more than 10 values. The dot in the notation only separates two components, so the syntax is *<ext>.<loc>* where *<ext>* is an integer representing the *external time* and *<loc>* is also an integer denoting the *local time* (so for example 1.0 < 1.5 < 1.10 < 1.15).

It is also worth mentioning that we can have deeper nesting of times than just two. If we wrote an event that waits for values from the `twice` event and produces two values in response, then the times of the produced values in reaction to the value 2000 would be something like 2.1.0 and 2.1.1.

Parallel merging. In the previous two examples, we used only `let!` (waiting for an event), `yield` (triggering an event) and `yield!` (to implement recursive loops). In this example, we look at an example that waits on multiple events using the `match!` construct. This feature can be used in two different ways. We can use `match!` to wait for the first of several events (as in section 2.5), which has a relatively clear semantics.

However, we can also use `match!` to wait for an occurrence of multiple events at once. This operation is similar to join patterns [31]. The main difference is that invocations on channels involved in a join pattern are buffered and join patterns can be used as synchronization primitive. In our library, `match!` joins first possible combination of event values starting from the time when it was called by the event-builder computation. We will define this meaning formally in section 5.2. The following example shows one case where the users would probably have a clear expectation about the result:

```
let oneTwoThree = event {
  let! _ = btn.MouseClick
  yield 1; yield 2; yield 3 }

let ticTacToe = event {
  let! _ = btn.MouseClick
  yield "tic"; yield "tac"; yield "toe" }

let merged =
  let rec loop() = event {
    match! oneTwoThree, ticTacToe with
    | !n, !s -> yield n, s; yield! loop() }
  loop()
```

The example defines two events that produce three constant values. Both of the event-builder computations that define them first wait on the same event (`btn.MouseClick`), which serves as a synchronization point to make sure that they will start producing values at the same time. If we use the nesting of times as discussed in the previous example and the first mouse click occurs at time 4, then the three values will be yielded with times 4.0, 4.1 and 4.2.

The merged event is constructed using a recursive loop that repeatedly waits for a value from the `oneTwoThree` and `ticTacToe` events and produces a single value (containing a tuple) in response. It seems reasonable to expect that when `oneTwoThree` and `ticTacToe` start producing values at the same time, the `merged` event will trigger three times carrying tuples with corresponding values from the source event. In our example, the synchronization is achieved by waiting for the same event before yielding, so the expected meaning of the `merged` event is the following:

*val merged : (time * (int * string)) list =*
[4.0.0, (1, "tic"); 4.1.0, (2, "tac"); 4.2.0, (3, "toe")]

The time in our library isn't discrete and based on steps as for example in [3], but isn't fully continuous as in [1]. This makes it possible to work with events that occur in parallel without committing to completely synchronized solution.

3.2 The structure of semi-discrete time

The time in our application has a very interesting structure. We can take any time value, turn it into a local time value (by adding ".0" to the end) and then increment the value arbitrarily many times without reaching a value that would be larger than the successor of the original value.

Technically speaking, the structure corresponds to a list of integers, so it is probably intuitively easy to understand. However, we can define the structure of the time more formally and use the following algebraic definition:

Time $(T, 0, succ, local, \leq)$ is a structure where:
- T is a set and $0 \in T$ is a selected element of the set
- $succ$ and $local$ are unary functions $T \to T$
- $\leq$ is a binary relation on T $(\leq \subseteq T \times T)$

The following axioms hold about ***time***:
- $\nexists t$ such that $succ(t) = 0$ or $local(t) = 0$ (Zero)
- $t \leq t$ and $t \leq succ(t)$ (Comparison)
- $local(t) \leq t$ and $t \leq local(t)$ (Weak equivalence)
- $succ^n\big(local(t)\big) \leq succ(t)$ for any $n \in N$ (Local time)

When working with time, we construct the time values from the initial time 0 (at the start of a reactive program) using the two provided functions. One important observation about the declaration is that the given axioms relate any two values (constructed from 0 by applying the functions *succ* and *local*) using the $\leq$ relation, which means that the set is fully ordered. Also note that 0 is not only the smallest element (as it is not a successor of any other time), but also a value of the most global time (as it is not a local time of any other value).

We will use the structure when calculating with time in the operational semantics. In the upcoming text, we'll use the following convenient notation instead of calling the *local* and *succ* functions explicitly:

$t + 1 = succ(t)$
$t.0 = local(t)$

Now that we have a precise definition of the structure of time, we can look at the formal semantics, starting with the syntax.

3.3 Syntax of reactive applications

Our reactive extension is implemented as an ordinary library, so we can use the full syntax provided by F#. The event-builder computation is a case of F# computation expression [28]. However we will define the semantics only for programs written using the pattern that we used in all the examples in this paper so far. The allowed subset of the F# syntax is shown in Figure 21.

prog = **let** ide_i = $event_i$ in *prog*	Program declaration
| *ide*	Start the main event
event = [ids_i = λ id_i . $eexpr_i$]	Event consisting of a list of
in *ids expr*	…states and an initial state
epat = !*id*	Wait for event pattern
| _	Ignore event pattern
eexpr = **let** *id* = *expr* **in** *eexpr*	Value binding
| **let!** *id* = *ide* **in** *eexp*	Waiting for an event
| **yield** *expr*	Triggering an event
| **yield!** *ids expr*	State transition
| *eexpr* ; *eexpr*	Composing computation
| **if** *expr* **then** $eexpr_1$	Conditional choice between
else $eexpr_2$	…event-comptuations
| **match!** *ide-list* **with**	Waiting for a combination
[$epat\text{-}list_i$ -> $eexpr_i$]	…of events (join pattern)
| ()	Empty computation

Figure 21. Simplified syntax of reactive programs

We define a program (*prog*) as a series of event declarations (an event declaration is for example `countDown`, `smileyMoves` and `smileyClicks` from section 4) followed by one identifier that specifies the main event of the program. This is slightly different to our implementation above, where we didn't have any main event and instead, the last event of the program performed some imperative side-effects. In the formal semantics, we can imagine that the main event yields instructions to the GUI as suggested in section 4.5. An event declaration (*event*) consists of an identifier, a series of functions that encode a state machine and a call to the function representing initial state of the state machine.

The most interesting part of the syntax is the event-builder expression, which specifies what constructs can be used to create events (in the F# implementation, this corresponds to the code inside the `event { … }` block).

Note that we use a different syntactic category for different kinds of identifiers. The category *id* is used for names of standard F# values and parameters. Functions that encode states of a state machine are from the category *ids* and finally, the names of events are from the *ide* category. This makes it possible to easily restrict what uses of `let!` and `yield!` are allowed. In F#, we can, of course, use any expressions, but in the semantics we add a few restrictions.

When performing a recursive call (or a state transition) using `yield!`, the parameter has to be an application of some state function that is declared as part of the event. In the syntax, the parameter is *ids expr*, which is an application of a function from the special category of state functions. Similarly, when waiting for an occurrence of an event using `let!`, we specify that the source should be an identifier from the category *ide*, instead of an expression. This restriction means that we can consume values only from previously declared events that are already running and that we cannot construct new events on the fly. The category *ide* also includes all external system events such as `btn.MouseDown` and others.

In the syntax definition, we use the *expr* category without defining it. This is the category of usual ML-like expressions as defined for example in [54]. We will also assume that we have the usual operational interpretation for ML expressions.

3.4 Semantics of event builder computations

We will not discuss static semantics of the described reactive language, because it is a subset of the full F# language and we use the typing rules of the host language. Events are standard F# values and have a type `Event<'a>` where `'a` is a type parameter specifying the type of values produced by the event when it is triggered.

Types and arrows. The dynamic semantics of our library is far more interesting. As already mentioned, our goal is to model events as a list of time/value pairs. The structure of the time was formally defined in section 5.2 and *value* denotes a set of values that can be evaluated as the result of standard ML (or F#) expression as defined in [54].

When specifying the meaning of an event-builder expression from the syntactic category *eexpr* (for example `yield 1`), we need a context that defines the time when the operation is executed and an environment that defines the meaning of identifiers that can appear in the expression. The environment is a disjoint union of three functions that provide the meaning of three types of identifiers:

$$\begin{aligned} env \;=\; & (id \to value) + \\ & (ids \to id \times eexpr) + \\ & (ide \to time \times [time \times value]) \end{aligned}$$

We write *env*(*id*), *env*(*ids*) and *env*(*ide*) to get values associated with individual identifiers. This is not ambiguous in any way as we always know the syntactic category of the identifier and these categories are disjoint.

The environment consists of three types of associations. Identifiers that represent values and parameters (*id*) are mapped to standard ML values. This models eager evaluation usual in the ML-family of languages. Identifiers representing state machine functions (*ids*) return the syntactic definition of the function, which gives us an easy way to deal with recursively defined functions. Note that the state machine function is always unary (which is not a limitation in practice as we can use tuples). Finally, the last function assigns meanings to identifiers representing events (*ide*). The meaning is the result of evaluating an event-builder computation. This is a pair that consists of a time specifying when the event-builder computation finished and a list of time/value pairs produced by running the computation.

Our operational semantics uses the following three types of reductions to specify the interpretation:

$$\begin{aligned} \to_e &: expr \to (env \to value) \\ \to_p &: prog \to (env \to [\, time \times value \,]) \\ \to_v &: event \to (env \to time \times [\, time \times value \,]) \\ \to &: eexpr \to (time \times env \to time \times [\, time \times value \,]) \end{aligned}$$

The function $\rightarrow_e$ interprets a standard ML expression and gives a function that returns a value when provided with an environment that assigns values to free variables (identifiers) occurring in the expression. We do not present the definition of this function as it is already provided in The Definition of Standard ML [54]. The rest of the functions will be defined in the next section.

The function $\rightarrow_p$ gives a meaning of the entire program. It takes an environment that consists of events defined externally (e.g. `btn.MouseDown`) and produces a list of time/value pairs that is generated by the main event of the program.

The function $\rightarrow_v$ defines a meaning of the *event* category, which is a definition of an event consisting of several mutually recursive function declarations. The result is a function that returns a completion time of the event together with a list of produced time/value pairs when given an environment. The environment is needed as it may contain definitions of other events used in the declaration (for example using the `let!` construct).

Finally, the most interesting function that our semantics defines is $\rightarrow$. It gives an interpretation of event-builder expressions such as `let!` and `yield`. The function takes a context consisting of a time when the expression starts evaluating and an environment that defines identifiers. As a result, it gives a time when the operation completes and a list of produced time/value pairs.

The rules that define operational semantics are shown in Figure 22. Rules *Program* and *Main* define the meaning of reactive program (syntactic category *prog*), the rule *Event* specifies the meaning of an individual event, which forms a part of the program. The rest of the rules specify meaning of the constructs that can be used in the event-builder expressions (category *eexpr*), with the exception of `match!` which will be discussed in section 3.5.

$$\frac{\begin{array}{c} env \vdash event \rightarrow_v \ t, e \\ e_{end} = [(t_{end}, ())] \quad t_{end} = t' \text{ such that } t' \geq t \ \& \ t' = 0 + 1 + \cdots + 1 \\ (env[ide \mapsto (t, e), ide.\text{Completed} \mapsto (t_{end} + 1, e_{end})]) \vdash prog \rightarrow_p r \end{array}}{env \vdash \textbf{let}\ ide = event\ \textbf{in}\ prog \rightarrow_p r} \ (Program)$$

$$\frac{\begin{array}{c} env_s = env \cup \{ids_1 \mapsto (id_1, eexpr_1), \ldots, ids_n \mapsto (id_n, eexpr_n)\} \\ id_{init}, eexpr_{init} = env_s(ids) \\ env \vdash expr \rightarrow_e v \quad (0, env_s[id_{init} \mapsto v]) \vdash eexpr_{init} \rightarrow e \end{array}}{env \vdash [ids_i = \lambda id_i . eexpr_i]\ \textbf{in}\ ids\ expr \rightarrow_v \ e} \ (Event)$$

$$\frac{env \vdash expr \rightarrow_e \textbf{true} \quad (t_0, env) \vdash eexpr_1 \rightarrow e}{(t_0, env) \vdash \textbf{if}\ expr\ \textbf{then}\ eexpr_1 \textbf{else}\ eexpr_2 \rightarrow e} \ (IfTrue)$$

$$\frac{env \vdash expr \rightarrow_e \textbf{false} \quad (t_0, env) \vdash eexpr_2 \rightarrow e}{(t_0, env) \vdash \textbf{if}\ expr\ \textbf{then}\ eexpr_1 \textbf{else}\ eexpr_2 \rightarrow e} \ (IfFalse)$$

$$\frac{(t_0, env) \vdash expr_1 \to t_1, vs_1 \quad (t_1, env) \vdash expr_2 \to t_2, vs_2}{(t_0, env) \vdash eexpr_1; eexpr_2 \to t_2, vs_1 @ vs_2} \ (Seq)$$

$$\frac{\begin{array}{c} s = env(ide) \\ (t, v) \in s \text{ such that } t = \min\{t' \mid (t', v') \in s;\ t' \geq t_0\} \\ (t.0, env[id \mapsto v]) \vdash eexp \to e \end{array}}{(t_0, env) \vdash \textbf{let!}\ id = ide\ \textbf{in}\ eexp \to e} \ (Waiting)$$

$$\frac{\nexists (t, v) \in env(ide) \text{ such that } t \geq t_0}{(t_0, env) \vdash \textbf{let!}\ id = ide\ \textbf{in}\ eexp \to \bot, []} \ (Stuck)$$

$$\frac{(t, l) = env(ide)}{env \vdash ide \to_p l} \ (Main)$$

$$\frac{env \vdash expr \to v \quad (t_0, env[id \mapsto v]) \vdash eexpr \to e}{(t_0, env) \vdash \textbf{let}\ id = expr\ \textbf{in}\ eexpr \to e} \ (Bind)$$

$$\frac{}{(t_0, env) \vdash () \to t_0, []} \ (Empty)$$

$$\frac{env \vdash expr \to_e v}{(t_0, env) \vdash \textbf{yield}\ expr \to t_0 + 1, [t_0, v]} \ (Trigger)$$

$$\frac{\begin{array}{c} id, eexpr = env(ids) \quad env \vdash expr \to_e v \\ (t_0, env[id \mapsto v]) \vdash eexpr \to e \end{array}}{(t_0, env) \vdash \textbf{yield!}\ ids\ expr \to e} \ (Transition)$$

Figure 22. Formal semantics of reactive applications

Semantics of program and events. The meaning of a reactive program is defined by two rules. The *Program* rule processes individual events that form the program and add two events to the environment before evaluating the rest of the program. The semantics defines the event "*ide*.Completed" which fires exactly once after the event named *ide* completes. This corresponds to the implementation of the `Completed` property discussed in section 2. The time of the `Completed` event occurrence is the most global time, so for example if the last value is produced at time 4.2.0, the `Completed` event will fire at time 5. The event produces a unit value, which is written as `()`. The *Main* rule simply gives the meaning of the specified main program event and discards the ending time, which isn't relevant for the semantics of the program as a whole.

It is worth mentioning that the semantics of *prog* doesn't allow recursively defined events. The rule *Program* adds the meanings of events to the environment in the order in which they are defined. If we attempted to recursively refer to some

future event, it wouldn't be defined in the environment. This is an intentional restriction of the library and it means that data-flow diagrams (such as the one presented in section 4.1) can't contain cycles.

When defining the meaning of an event in the Event rule, we construct an environment envs, which extends the original environment with definitions of individual state functions. Then the rule evaluates the meaning of the initial state function starting at the time 0. The resulting completion time and a list of values is the meaning of the event.

Semantics of event-builder. First of all, the rules *IfTrue*, *IfFalse* and *Bind* do not specify any special behavior with respect to time or events. They define the meaning simply in terms of another event-builder expression, possibly with an extended environment, so we won't discuss them in details.

From the remaining rules, Empty is the simplest one. It specifies that an empty event doesn't produce any values and completes running at the same time at which it was started. In practice, an empty event is used implicitly in an `if` expression that doesn't contain the `else` clause. In the *Seq* rule, we simply evaluate both of the event-builder computations in a sequence and concatenate the produced values using the @ operator.

The *Waiting* rule is more interesting. It obtains the meaning of the event that it is supposed to wait for from the environment and then selects the first event that occurs after or at the time t_0 (the time when the computation was started). This is done using the *min* function which selects the minimal time from a given set and could be defined as follows:

$$\min(s) = \ t \in s \text{ such that } \forall t' \in s : t' \geq t$$

The definition is valid, because (as noted in section 5.2) it is possible to compare any two *time* values. The event-builder expression that follows `let!` is started with the initial time set to *t.0*, which means that the execution will use a local time relatively to the time of the handled event occurrence. This guarantees that all events that are produced in reaction to the event will occur before another the source event (*ide*) occurs again and so the user will be able to handle all occurrences using a recursive loop. Note that the *Waiting* rule cannot be used if the event we're waiting for never occurs in the future. This case is handled by the *Stuck* rule, which returns an empty list and an undefined ending time.

The *Trigger* rule specifies that when `yield` runs at some specified time, it produces a single value at that specified time. It is worth noting that the returned time of completion is incremented by one. Together with waiting, this is the only operation that takes some time and as discussed in section 5.1, this is needed in order to distinguish values produced by subsequent yields. Finally, the *Transition* rule obtains a value that should be passed as an argument to the state function from the environment and evaluates the body of the state function.

3.5 Semantics of the match! construct

Specifying the meaning of the `match!` construct is slightly more complicated. It needs to wait for several combinations (or joins) of events that are created by

individual clauses and select the first clause for which the values become available. Moreover, one event of the join may occur multiple times while waiting for another event of the join, so we need to specify which of the values should be selected.

In the implementation, the clauses of match! can also contain nested patterns (which specify that only some of the produced values are accepted) and the `when` clause which allows us to specify arbitrary conditions on the values. In the formal semantics, we don't describe these two features, because that would increase complexity and make the key idea less visible.

Waiting for events. Each clause of the `match!` construct constitutes a single join that waits for several events in parallel. We start by discussing how this waiting works. The behavior can be best demonstrated graphically and you can see it in Figure 23. The top three horizontal lines represent source events and circles, squares and triangles represent values produced by the events. The dashed horizontal line represents the result and finally, the line marked t_0 denotes a starting time of the waiting.

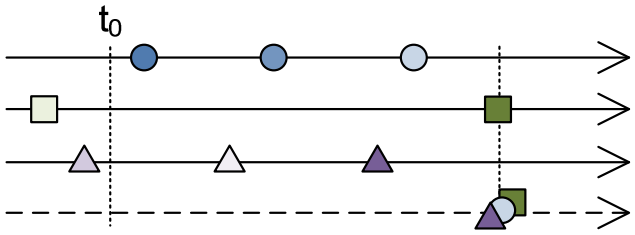


Figure 23. Waiting for multiple events

Note that the result isn't actually an event as we need only a single combination of values (so that we can resume the computation by running the body of the clause). This is quite different to operations that combine events occurring possibly multiple times into a new event that also occurs possibly repeatedly (such as the `SelectMany`, `CombineLatest` and `Zip` operations in [9]).

As the diagram demonstrates, we ignore values that may have been produced before the starting time and we wait until all of the events produce at least one value. The time we return as part of the result is the time of the last event that we were waiting for. Now, there are several ways to choose the values to be returned:

- **First values.** In this approach, we could select the first value that was triggered by each of the events.
- **Last values.** This way we select the last value that was triggered by an event before the last event that we were waiting for finally fires.

In our reactive library, we use the second technique for two reasons. We find it more practically useful – for example when waiting for values from all available sensory inputs in a robot (such as temperature sensor, gyroscope and GPS locator), we are more likely interested in the most recent value. The second reason is that the *first-value* semantics can be easily simulated, while simulating the *last-value* semantics using *first-value* implementation isn't easily possible. The function to do this can be written as follows:

```
let first(evt) = event {
  let! val = evt in yield val }

match! first e1, first e2 with
| !v1, !v2 -> // computation body
```

When the function first runs (immediately before `match!` is processed) it waits for the first event and triggers the returned event with this value as an argument. All subsequent occurrences of the event are ignored. Although the formal semantics doesn't allow writing functions that take events as parameters, this is perfectly possible in the actual implementation.

Formal definition. The formal definition of the `match!` construct is expressed using two rules defined in **Figure 24**. We define an auxiliary function that specifies waiting for multiple events as discussed in the previous operation:

$$\rightarrow_{\mathrm{w}} : \ [\, id \times ide \,] \rightarrow (time \times env \rightarrow time \times env)$$

The first argument of the function is a list of tuples consisting of two identifiers. The second identifier (*ide*) is a name of an event that we want to wait for and the first identifier specifies the name of a variable where we want to store the result of waiting. The context (specified as an argument of the returned function) consists of a time where we want to start waiting and an environment, which contains (among other things) definitions of events that we want to wait for. As a result, the function gives us a time when the last of the awaited events occurred and an environment that assigns values carried by the events to the identifiers specified in the first parameter.

The function is defined by *Join* and *JoinStuck* rules. The context of the rules provides an environment and the starting time. We first define t_{min}, which is the smallest time when a value becomes available for all events. Then we collect last produced values (as described in the previous section) and constitute a new environment to be returned as the result. Similarly to *Stuck*, the *JoinStuck* rule provides a case returning an undefined time and an empty list of time/value pairs when there are no further occurrences of the join.

The *Match* rule defines the meaning of the → function for the `match!` construct which was missing in Figure 24. For each clause, we first select all patterns (the *epat* syntactic category) specifying that we want to wait for an event together with the associated event. Next we use $\rightarrow_{\mathrm{w}}$ to obtain the time when the clause could run and the environment with values carried by the events. Finally, we select the first clause and interpret the body of the clause under a context consisting of a time and an environment constructed as a union of the original environment with an environment assigning values to the variables occuring in the clause patterns. Again, we also provide a rule for the case when there are no furhter occurrences (named *MatchStuck*).

$$\frac{\begin{array}{c}(\mathrm{t}_0, env) \vdash \{(id, ide) \mid (epat, ide) \in \{(epat_{i,1}, ide_1), \ldots, (epat_{i,n}, ide_n)\} \text{ such that } epat = !\, id\ \} \rightarrow_w t_i, env_i \\ (t, env_{vars}, eexpr) \in \{(t_1, env_1, eexpr_1), \ldots, (t_k, env_k, eexpr_k)\} \text{such that } t \neq \bot\ \&\ t = \min\{t_1, \ldots, t_k\} \\ (t.0, env \cup env_{vars}) \vdash eexp \rightarrow e\end{array}}{\begin{array}{l}\textbf{match}!\ ide_1, \ldots, ide_n\ \textbf{with} \\ (t_0, env) \vdash \ \mid epat_{1,1}, \ldots, epat_{1,\mathrm{n}} \rightarrow eexpr_1\ \ \rightarrow e \\ \quad\ \mid epat_{\mathrm{k},1}, \ldots, epat_{\mathrm{k,n}} \rightarrow eexpr_{\mathrm{k}}\end{array}}\ (Match)$$

$$\frac{\begin{array}{c}(\mathrm{t}_0, env) \vdash \{(id, ide) \mid (epat, ide) \in \{(epat_{i,1}, ide_1), \ldots, (epat_{i,n}, ide_n)\} \text{ such that } epat = !\, id\ \} \rightarrow_w t_i, env_i \\ \forall i \in \{1 \ldots k\}: t_i = \bot\end{array}}{\begin{array}{l}\textbf{match}!\ ide_1, \ldots, ide_n\ \textbf{with} \\ (t_0, env) \vdash \ \mid epat_{1,1}, \ldots, epat_{1,\mathrm{n}} \rightarrow eexpr_1\ \ \rightarrow \bot, [] \\ \quad\ \mid epat_{\mathrm{k},1}, \ldots, epat_{\mathrm{k,n}} \rightarrow eexpr_{\mathrm{k}}\end{array}}\ (MatchStuck)$$

$$\frac{\begin{array}{c}s_i = env(ide_i) \qquad t_{min} = \min\{\, t \mid (t, v) \in \cup\, \mathrm{s_i}\ \&\ \forall i \in \{1 \ldots \mathrm{n}\} : \exists (t', v') \in \mathrm{s_i} : t' \leq t_{min}\ \&\ t' \geq t_0\} \\ (t_i, v_i) \in s_i \text{ such that } t_i = \max\{t' \mid (t', v') \in s_i;\ t' \leq t_{min}\}\end{array}}{(\mathrm{t}_0, env) \vdash \{(id_1, ide_1), \ldots, (id_n, ide_n)\} \rightarrow_w t_{min}, [id_1 \longmapsto v_1, \ldots, id_n \longmapsto v_n]}\ (Join)$$

$$\frac{s_i = env(ide_i) \qquad \exists i \in \{1 \ldots n\}: \nexists (t, v) \in s_i \text{ such that } t \geq t_0}{(\mathrm{t}_0, env) \vdash \{(id_1, ide_1), \ldots, (id_n, ide_n)\} \rightarrow_w \bot, []}\ (JoinStuck)$$

Figure 24. Formal semantics of the `match!` construct

4. Guarantees

The main purpose of the formal semantics presented in the previous section is to provide a framework for reasoning about reactive applications written using our library. In this section, we show several properties about frequent usage patterns.

4.1 Simple recursive loops

In the section 1.4, we created an event that collects all drawn rectangles into a list and fires each time a new rectangle is created, carrying a list of all rectangles drawn so far. We implemented the functionality as a recursive loop that repeatedly waits for an event (using `let!`) and then reacts (using `yield`).

To show that the implementation is correct, we need to guarantee that the computation will handle all values produced by the source event. The following example implements an event that reproduces behavior of another event:

```
1: let e' =
2:   let rec loop () = event {
3:     let! v = e
4:     yield v
5:     yield! loop () }
6:   loop ()
```

Whenever the event `e` triggers a value (line 3), the event we create fires with the same value as an argument (line 4) and then recursively loops to wait for the next occurrence (line 5). The event `e'` should not only handle all occurrences of `e`, but it should also produce the same values. Since triggering the event using `yield` is the first reaction in the computation, we can also shows interesting fact about the time of the produced events. Let's now look at the formal semantics of a program that defines an arbitrary event `e` and the event `e'` shown above. The meanings of the two events are two ending times and two lists of time/value pairs:

$$env(e) = t, [(t_1, v_1), \ldots, (t_n, v_n)]$$
$$env(e') = t', [(t'_1, v'_1), \ldots, (t'_m, v'_m)]$$

We start by analyzing the two produced lists and we'll discuss the ending times later. First of all, we want to show that the number of produced elements is the same and that the values are the same and are produced in the same order. We can also reason about the time. If the time of the original event occurrence is for example 4.2, then the time of the reaction will be 4.2.0.

We say that two events are *weakly equivalent* if:

$$(n = m)\ \&\ \forall i\colon (v_i = v'_i\ \ \&\ \ t'_i \geq t_i\ \&\ t'_i \leq t_i)$$

Now, we can show that the events `e` and `e'` are *weakly equivalent*.

Proof. We say that an event is *sequential* if for each two consecutive occurrences (t_i, v_i) and (t_{i+1}, v_{i+1}), it is true that the time $t_{i+1} \geq t_i + 1$. We shall require that all external events (such as `btn.MouseDown`) are *sequential* and as a consequence we can show that all user-defined events are *sequential* too:

In the rule *Trigger*, the time of the occurrence is t_0 and the ending time of the constructed event is $t_0 + 1$ where t_0 is the starting time. From the rule *Seq*, we

can see that if there is any subsequent computation, it will not run before the time $t_0 + 1$. As there is no rule that would allow running computation at an earlier time than at the starting time, we can see that all events are *sequential*.

Now that we know that the event `e` must be *sequential*, we can show that the equation above holds:

Let's say that the event `e` occurs at time t. From the rule *Waiting*, we know that the line 4 will be executed at time $t.0$. The value `v` will be triggered at time $t.0$ (*Trigger*) and the line 5 will run at time $t.0 + 1$ (*Seq*). After the recursive call, the line 3 will run at time $t.0 + 1$. Thanks to sequentiality, `e` will not occur again before $t + 1$ and from the time axioms, we know that $t.0 + 1 \leq t + 1$.

The times of values produced by the event `e'` are not exactly the same (since $t.0$ is a different time value than t), but they are equivalent for many practical purposes. Next, we'll analyze the ending times of the two events (t and t'), No matter what the value of t is, the value of t' will always be $\perp$, because the event `e'` never stops running (after processing all values of `e`, the rule *Stuck* will be used).

4.2 Reasoning about pattern matching

In the previous section we looked at recursive processing of single source event using the `let!` construct. Now, we'll look at a more complicated scenario when we need to handle multiple source events and we need to use the `match!` construct (discussed in section 5.5). The following example event uses match! to wait for the occurrences of events `a` and `b`. When one of them occurs, it produces a value of type `Choice<'a, 'b>`, which contains the value with a tag that specifies which of the events generated the value:

```
1: let c =
2:   let rec loop () = event {
3:     match! a, b with
4:     | !v, _ -> yield Choice1Of2(v)
5:                yield! loop ()
6:     | _, !v -> yield Choice2Of2(v)
6:                yield! loop () }
7:   loop ()
```

The example is very similar to the one presented in section 6.1. The difference is that we now have two branches for handling two different events. Ideally, we'd like to show that if we take the meaning of `c` and take only occurrences tagged with `Choice1Of2`, we'll get an event that is *weakly equivalent* to the event `a` (in terms of the definition from section 6.1) and similarly for the event `b`. Unfortunatelly, this is not always the case.

Definition. To reason about the previous example, we need to define a *rank* of time value t. *Rank* is defined recursively by the construction of the time value:

$$rank(0) = 0$$
$$rank\big(local(n)\big) = rank(n) + 1$$
$$rank\big(succ(n)\big) = rank(n)$$

Intuitively, *rank* specifies the number of dots in the notation we use through this chapter. For example $rank(4.2) = 1$ and $rank(4.2.0) = 2$. In addition to rank of time, we also define a rank of an event. An event `e` has rank n if the times of all occurrences of the event have *rank* n.

Events of equal rank. A useful special case is when the two (or more) events that are being processed using `match!` have the same rank. This is the case for example in section 4.2 since all external events have rank 0 and the `Completed` event (triggered when an event stops producing values) also has rank 0.

If the two source events never occur at the same time, we get a result that is similar to the one from section 6.1. We will not show a full formal proof as it is very similar to the one presented earlier. When one of the events occurs at time t, we know that the time of the next occurrence of any of the two events will be at least $t + 1$ (the other event didn't occur at time t and since they have the same rank, it cannot occur between t and $t + 1$). Just like in the earlier proof, the reaction ends in time $t.0 + 1$, so the `match!` is called recursively before any of the events occurs.

General case. In the general case, we cannot guarantee that the recursive loop will handle both of the events. If the event `a` triggers at time t, then the processing will complete in time $t.0 + 1$, however the event `b` may be triggered during the processing (for example at time $t.0.0 + 1$). If both of the events have a *rank* (meaning that they trigger values at times with the same rank) then we can at least guarantee that no value of the event with the lower rank will be missed. Let's assume that the $rank(a) = n$ and $rank(b) = m$ such that $n \leq m$.

- Informally, if the event `a` fires, then the reasoning is the same as in section 6.1. If the event b fires at time t, then we need to show that it will finish processing before the other event can fire.
- Let t' be the maximal time of rank n smaller than t (that is, the last time at which `a` may have occured before `b` occured). Now, the event `a` can't trigger before time $t' + 1$.
- The time t has a higher rank than t', meaning that:

  $$t' + 1 \geq t.0 + 1$$

  As a result the next possible occurrence of the event `a` would be handled.

As we can see, the reasoning about `match!` is more complicated, but is certainly possible and may give useful results. As discussed in the future work, it should be possible to implement a compiler extension that would warn users about possible unhandled events in recursive processing loops.

4.3 Handling completion of events

In our last example of formal reasoning, we'll get back to the event that reproduces the behavior of another event. The implementation from section 6.1 was faithful in terms of generated values, but didn't correctly handle the completion of the event. To implement an event that ends in the same time as the source event, we need to handle the `Completed` notification:

```
1: let e' =
2:   let rec loop () = event {
3:     match! e, e.Completed with
4:     | _, !() -> ()
5:     | !v, _ -> yield v
6:                yield! loop () }
7:   loop ()
```

In this case we're matching on two events. The `Completed` event has rank 0 (as we can see from the *Program* rule) and the event `e` may not have a rank at all. This means that we cannot effectively use any of the conclusions from section 6.2.

However, the `Completed` event is very special. The *Program* rule essentially defines that it will occur once at a time, which is a least time of rank 0 after the last occurrence of `e`. Using similar reasoning as in the previous examples, we can show that the events `e` and `e'` are weakly equivalent. We cannot miss a value triggered by `e`, because `e.Completed` will not occur before all values are handled. However, we'd also like to show some relation between t (the ending time of `e`) and t' (the ending time of `e'`). The times are not equivalent, but if we follow the rules that define the ending time of an event, we obtain the following equation:

$$t' = (\min\{t_n \mid rank(t_n) = 0 \,\&\, t_n \leq t\} + 1).0$$

Intuitively, this means that the event `e'` will end "reasonably soon" after the event `e`. Note that if we modified the rule *Program* to trigger the `Completed` event immediately after an event ends, it would be difficult to prove that the event is not lost in any case. We explicitly made sure that `Completed` is an event of rank 0, which means that it will always be handled correctly by a recursive loop that uses `match!` (which follows from the discussion in section 4.2). This is very important in practice, especially when implementing hierarchical state machines.

5. Abstract imperative computations

When introducing F# computation expressions in section 1, we demonstrated how to use them for encoding standard types of computations such as *monads* and computations that correspond to the Haskell's `MonadPlus` type class. However, we also noted that our use of computation expressions doesn't strictly follow any of these two patterns. Although we use the same operations as `MonadPlus` type, our implementation does not obey the usual laws of the type.

In this section, we look at our `event` computation from the abstract point of view and we'll describe an alternative set of laws that our computation obeys. We show that this is a common pattern of inherently imperative computations that may be applicable to numerous other scenarios. We start by looking at another example of imperative computation that shares the same abstract properties as our `event` type, but is simpler.

5.1 Processing sequences in F#

When programming with collections or sequences of values in F#, we can use the `seq` computation which is similar to the `List` monad in Haskell. The *bind* operation of this type runs the rest of the computation for all elements of the sequence given as an argument and collects the results. For example a simple combination of filtering an projection may be implement like this:

```
> seq { let! num = [0 .. 9]
        if num%3 = 0 then
          return num*num }
val it : seq<int> = [9; 36; 81]
```

The computation gradually assigns a number from the input list to the symbol `num` and runs the rest of the computation, which may produce a singleton list (using `return`) or an empty list (implicitly in the `else` branch). In reality, F# uses slightly different notation (`for` instead of `let!` and `yield` instead of `return`), but the meaning is the same. This computation would be a valid instance of `MonadPlus` and obeys all the usual laws.

This syntax is somewhat limited. For example, if we use multiple subsequent `let!` constructs, the computation will behave as Cartesian product, which makes it difficult to implement for example the `zip` function. The type that represents sequences in F# is imperative (as it is a standard type used by the .NET Framework) and is defined as follows:

```
type IEnumerable<'a> =
  abstract GetEnumerator : unit -> IEnumerator<'a>
```

The `IEnumerator<'a>` type is an implementation of the Iterator pattern [55] and allows sequential iteration over the collection. When implementing operations where we need to take elements from multiple sources in parallel (such as `zip` or `merge`), we need to use the `IEnumerator<'a>` type and resort to the C#-like imperative style.

Computation for iterators. However, it is also possible to use computation expressions for providing an elegant syntax for imperative processing of the `IEnumerator<'a>` type. A slightly simplified definition of the type looks as follows:

```
type IEnumerator<'a> =
  abstract Current : 'a
  abstract MoveNext : unit -> bool
```

When working with the type, we use the `Current` property to read the current element and the `MoveNext` method to navigate to the next element (until the method returns `false`).

Now, we can define an imperative computation for working with enumerators. It constructs a new `IEnumerator<'b>` value and we can use yield to produce next element to be returned. More interestingly, we provide a `let!` operation that takes a single element from the `IEnumerator<'a>` given as an argument and runs the rest of the computation with this value as an argument. If a value is not available, the computation ends. Let's look at a simple demonstration:

```
1: let nums = iter {
2:   yield 1; yield 2 }
3: let rest = iter {
4:   let! n = nums
5:   yield "first: " + n.ToString()
6:   let! m = nums
7:   yield "second: " + m.ToString()
8:   let! k = nums
9:   yield "second: " + k.ToString() }
```

If we iterate over all the elements available in the `rest` value, we get elements "first: 1" and "second: 2". Then will the iterator end (returning `false` from `MoveNext`). Even though we're working with ordinary iterators, the example shares many properties with our reactive library for working with events.

The most important similarity is that the `let!` operation has some side-effect and we need to apply it repeatedly on a value representing some computation to get all values produced by the computation. If we want to process all elements, we need to do that using a recursive loop. The similarity will be even more obvious if we look at the following implementation of the `zip` function for `IEnumerator<'a>` values:

```
let rec zip xs ys = iter {
  match! xs, ys with
  | x, y -> yield x, y
            yield! zip xs ys }
```

The recursive processing pattern is similar to the one we would write when implementing `zip` for immutable functional lists. The key difference is that binding on `IEnumerator<'a>` is an operation that modifies the object, so we pass the same object as an argument to the recursive call. In the functional implementation, binding gives us a new object representing the rest of the list.

As already mentioned, the `iter` computation introduced in this section shares many properties with our reactive library and we tend to use similar patterns such as processing using recursive loops. In the next section, we describe this type of computations abstractly and relate it to the *monad* and `MonadPlus` types.

5.2 Monad laws

As discussed in the introduction, F# allows us to use either the `return`/`return!` notation (usually preferred for standard monads) or `yield`/`yield!` notation (used especially by various forms of sequences or `MonadPlus` computations). Our reactive library as well as the previous toy example used the second notation, so we'll continue using it, even though it may feel unfamiliar when discussing monads. The **Figure 25** uses the F# notation to present the usual laws that are supposed to hold about monads.

```
m { let! x' = m { yield x }
    yield! f x }                ≡  f x                          (Left identity)

m { let! y = m { let! x = m
                 yield! f x }   ≡  m { let! x = m
    yield! g y }                       let! y = f x             (Associativity)
                                       yield! g y }

m { let! x = m
    yield x }                   ≡  m                            (Right identity)
```

Figure 25. Monad laws written using the computation expression syntax

If we analyze the laws, we can see that the first two laws hold for our two examples of imperative computations as well. For events, we can prove this formally using the weak equivalence defined in section 4. For iterators, this would require defining a formal semantics, but it isn't difficult to verify the laws informally.

The *Right identity* law however fails to hold for both of our computations. The reason is that in both of the cases, `let!` runs the rest of the computation only once even though the monadic value `m` may (eventually) produce multiple values. To implement a computation that behaves identically to the source computation, we need to implement a recursive processing loop. We will look at alternative formulation of the law for our imperative computations in the next section.

There are also some laws that should hold for computations represented as the `MonadPlus` type. These are less widely accepted. However, from the laws defined in [56], our computations obey both of the more generally accepted laws – namely the *Monoid* law and the *Left zero* law.

5.3 Imperative computation laws

Our imperative computations do not obey the *Right identity* law, but they have a property that is very similar to this law. We can construct an equivalent computation by writing a recursive loop that processes values one by one. To describe this property formally using an equation, we need to use a fixed point combinator, which makes it possible to declare a recursive function without naming it using the `let rec` keyword. In F#, a fixed point operator can be defined as follows:

```
> let rec fix f x = f (fix f) x;;
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

The `fix` function can be used for defining recursive functions that take some argument of type `'a` and produce result of type `'b`. The function given as the first argument will be called with two parameters – the second one is the input and the first one is a function that can be used to make the recursive call. The following example shows how to use the `fix` function to encode a simple recursive function such as *factorial*:

```
> let fact = fix (fun f x ->
    if x <= 1 then 1 else x * (f (x - 1)));;
val fact : int -> int
```

The first parameter of the lambda function is a function that can be used for the recursive call and we use it in the `else` branch to calculate factorial of *x - 1*.

Recursive identity. Now, we'll look how to encode the usual recursive loop used for processing imperative computations. We want to use the `fix` combinator to encode a loop like the one presented in section 6.1. This construction should create a computation that is equivalent to the one we're getting values from (using `let!`), so we can use it as an alternative to the usual *Right identity* law.

Let's say that `comp` is some computation of type `M<'a>` for which we have a computation builder `m` that supports `let!`, `yield` and `yield!` operations. We can encode a recursive function that uses `let!` once and then calls itself recursively using `yield!` as follows:

```
fix (fun loop () -> m {
  let! x = comp
  yield x
  yield! loop () }) ()
```

The `loop` value that we get as a parameter from `fix` is a function that takes unit value as an argument and runs the computation once to read and `yield` the next element. In the example for working with `IEnumerator<'a>`, the constructed enumerator will be exactly the same as `comp`. In our reactive library the computations will be *weakly equivalent* (Section 4.1). We could also define the event computation in a way such that it would automatically end when the event (or events) that we're waiting for (using `let!` or using `match!`) ends. This way the computations would be equivalent in terms of the definition from section 4.3. However, for practical reasons, we prefer the first option and we find that *weak equivalence* is powerful enough.

Simplified recursive identity law. We will make one more simplification to present the law in a more accessible form. We will declare the computation as a recursive value rather than a recursive function that calls itself. This definition can be implemented only in a language with lazy evaluation, but we use it mainly for a formal presentation.

We assume that the computation `M<'a>` provides following functions, which correspond to `let!`, composition of computation expressions and the yield keyword. Note that there is no special operation for encoding the `yield!` construct. This is because this construct translates (essentially) to an identity function. The types of the operations are follows:

```
>>=     : M<'a> -> ('a -> M<'b>) -> M<'b>
⊕       : M<'a> -> M<'a> -> M<'a>
return  : 'a -> M<'a>
```

These are 3 of 4 standard operations required by the `MonadPlus` type class. In addition, we also need a fixed-point operator, this time for encoding value recursion:

```
fix :('a -> 'a) -> 'a
```

The operator takes a function that declares a value of type `'a` in terms of itself (given as an argument). In a lazy language, this can give a perfectly valid definition. Now, we can encode the loop for processing computations as a recursive computation of type M<'a> that composes code for handling a single value with the computation itself. The following equation describes *Recursive identity* law, which is our alternative to the monadic *Right identity*:

$$m \;=\; fix\,(\backslash r \rightarrow m \gg\!= (\backslash x \rightarrow (return\; x) \oplus r))$$

The law describes essentially the same construction as the previous code, but is written explicitly using the operations provided by the `MonadPlus` type class. Our imperative computations cam be now abstractly described as computations that support operations provided by the `MonadPlus` type class and obey the *Left identity* and *Associativity laws* (inspired by monads), the *Monoid* and *Left zero* laws (inspired by `MonadPlus`) and the *Recursive identity* law. To support the `match!` construct, we also need to provide the two operations described in Chapter V.

Chapter VII

Ideas for future work

The work presented in this thesis spans several interesting areas ranging from garbage collection to programming models and reasoning about programs. In this chapter we present several ideas that are motivated by our work and that would extend the work in directions that we consider as very interesting.

The reactive programming model has been an active research topic in the academic community for a long time, however it is still used in relatively specific areas such as embedded systems. We believe that there is a great potential for this programming model nowadays. The user interfaces are becoming more interactive and a modern application needs to communicate with numerous data sources, which is best done asynchronously. This raises the question whether we could provide a direct support for reactive scenario in garbage collectors as well, not directly for our programming model, but in general.

Another present trend in the theory of programming languages is automatic verification of systems and the aim to provide stronger guarantees about programs. We can follow this direction in several places of our work. Firstly, we'll discuss whether the compiler could check some basic properties of the `match!` construct. Secondly, we demonstrated how to formally reason about reactive applications implemented using our programming model, but we could ask if some of the reasoning could be done automatically?

1. Garbage collection in reactive scenario

In the Chapter IV, we have focused on the definition of garbage and finding an implementation that doesn't cause memory leaks in the situations when we combine declarative and imperative style of reactive programming as introduced in the introduction in Chapter II.

However, similar problems have been observed for other programming models such as the actor model [15]. This suggests that we may need more powerful GC algorithms for reactive programming in general. It is not yet clear to us whether the work on liveness analysis [22, 23] can provide a more general solution. The algorithm we proposed serves mainly as a useful formal model for the design of reactive library, but it would be interesting to see if it can be generalized to cover all known reactive scenarios and implemented in practice.

Another problem is to show that garbage collection in the reactive (or more generally, in any non-standard) memory model in fact needs a specialized GC algorithm. In section 4.6 of Chapter IV, we discussed why we cannot use weak references in our model, but there are other advanced features such as ephemerons [26] that may provide the necessary expressive power. In general, this demonstrates the need for a solid framework for formal reasoning about the expressivity of garbage collection techniques.

2. Compile time checking for match!

In section 7.4 of Chapter V we draw a distinction between two types of monads – those that always eventually produce a value and those that may (in some cases) never produce value. In this section, we briefly discuss the compile-time checking that could be provided for these two types.

2.1 Values available

For the first type of monad (e.g. `RandomMonad` or `Future`), we know that there will eventually be a value available for every binding pattern. The ignore pattern may make it possible to run the clause earlier (e.g. before a future finishes), but otherwise behaves similarly to standard underscore pattern. As a result, we can transform our *computation patterns* into usual *patterns* and then run the standard incompleteness check. The transformation follows two simple rules:

!<*pat*> → <*pat*> (binding pat.) _ → _ (ignore pat.)

The first rule extracts the underlying *pattern* of a *binding pattern*. The second rule transforms an *ignore pattern* written as "_" into a standard *underscore pattern* (which is incidentally also written as "_"). It is worth noting that the *choose* operation which selects clauses may have some reasonable notion of default behavior for a case when values don't match any clause (such as returning `None` in the `Maybe` monad). Ideally, the author of *choose* should be able to specify the required behavior of incompleteness checking.

2.2 Missing values

When using pattern matching with monadic values that may never produce an actual value, we need to consider if the matching can succeed when some values are not available. Firstly, there may not be any value available at all. In this case, the *choose* operation should behave the same way as *bind* when matching on a single monadic value, which doesn't contain an actual value. We will discuss this problem in more details shortly.

To cover all other cases, it is necessary and sufficient to provide a set of clauses that (together) form a complete pattern for each of the arguments and don't contain any other *binding pattern*. The following example shows a complete pattern matching in the `Maybe` monad:

```
match! opt1, opt2 with
| _, !_       -> "second"
| !(x::xs), _ -> "first - cons"
| ![], _      -> "first - nil"
```

The first clause contains a complete pattern for all possible values of the second argument. The first argument is a list, so the last two clauses form a complete pattern for the first value. In case when all values may be missing, the *choose* operation should have some default behavior. Depending on the monad, it may or may not be desirable to warn the user about incomplete patterns.

2.3 Ignore all pattern

The translation semantics from Figure 12 does not allow the case when a clause consists solely of *ignore patterns*. In practice, this doesn't appear to be a limitation for monads that always produce a value. When the time is involved (e.g. `Future`), the "ignore all" clause could be always invoked (although this would be non-deterministic). When time is not involved (e.g. `RandomMonad`), the ignore all clause is equivalent to a clause consisting of binding patterns "!_".

However, the "ignore all" clause seems to be useful for monads that may not produce a value. For example, in the `Maybe` monad, we would expect that the "_, _" clause matches `None`, `None` case. Even though this may sound intuitive, there isn't any reasonable way to express it. The problem is that checking for absence of value is an operation that can be done only outside of the monad. A *binding pattern* specifies binding inside the monad, but an *ignore pattern* gives only a negative statement – we don't need to bind on a particular value. All other clauses are constructed from positive statements (bind on some value). We believe that the default behavior of choose is the right one in most of the cases, however finding a way to encode the "ignore all" patterns is an interesting future problem.

3. Committing monadic computations

Most of the monadic values can be accessed via multiple references from the program without any need for synchronization. This is obviously true for all immutable values, but it is also true for some mutable monadic value (e.g. futures).

However, we also want to work with monadic values that require some synchronization when a clause is selected by the *choose* operation. A typical example is the `join` builder that we use for encoding join calculus. The computation is based on channels. Outside of the monad, we can send messages to channels and the monad allows us to read them. However, each message can be received only by one computation.

This behavior can be implemented using the combinators we presented. Roughly, the channels composed using *merge* need to reference the original source channels and the *choose* operation needs to perform a commit operation when it selects a clause. At this point it takes the value out of the channel (a channel constructed using *merge* removes values from their original source channels at this point) and cancels all other clauses.

Note that the computation runs in two different *modes*. In one mode, channels are protected and we can only send or receive a single message. In the second mode, the pattern matching is in progress and we needs to access messages in the channel directly. When implementing this kind of monad, we need to encapsulate both computation modes inside a single type, which makes the code more complicated.

If we wanted to support the "two mode" style of computations more directly, we could generalize the type of our combinators to work with two types. The type `M<'a>` represents standard monadic value; the type `MA<'a>` represents a value when pattern matching is in progress (alternative mode):

```
val ⓘ      : MA<'a> -> M<'b> -> MA<'a * 'b>
val map    : ('a -> 'b) -> MA<'a> -> MA<'b>
val maunit : MA<unit>
```

The modified version of the merge operation together with the maunit value allows us to combine multiple (standard) monadic values into a single value in the alternative mode. We can for example write `(((maunit ⓘ a) ⓘ b) ⓘ c)` to combine three monadic values (the additional unit value can be automatically dropped later). We also need the `map` function for computations in the alternative mode.

Now, the *choose* operation needs to be modified to take a list of computations that represent pattern matching in progress. It selects a clause, performs the commit operation and then returns the body of the clause:

```
val choose : list<MA<MaybeDelayed<M<'a>>> -> M<'a>
```

In this thesis we used the simpler version, because we believe that it is sufficient in most of the common situations and it is also clearer when studied formally. However, we consider "two mode" style of computations an interesting and potentially useful generalization of our work.

4. Automatic verification of reactive programs

Automatic program verification has recently some very promising results. In this thesis we presented a formal model for reasoning about reactive programs developed using our reactive programming model. We used this formalism to show that, under some conditions, an occurrence of an event will not be missed during imperative processing. This kind of property is very important for reactive programs, so it would be desirable to automate this verification.

This is a very challenging area, but the recent results are encouraging. More specifically, automated program verification could be used to verify the following properties of reactive programs.

Missing events. A property that we need to verify the most frequently is that a certain computation will not miss any occurrence of an event (assuming that code executed in response to some event terminates). This can be relatively easily verified for recursive loops constructed using the `let!` construct. Implementing an extension for the F# compiler that would automatically verify this property would not require sophisticated program verification techniques.

When implementing a recursive loop using the match! construct, the task is somewhat more complicated. As demonstrated in section 4.2 of Chapter VI, we need to analyze the rank of an event. This analysis is also relatively straightforward. However, it would be beneficial to provide the user with some way for specifying which events should not be missed, so that the compiler doesn't show warnings in cases where the missing of an event is intentional (or doesn't matter).

Liveness in general. A recent work by Cook et al. [66] shows that it is possible to "prove that programs eventually do something good", meaning that we can use automated tools to prove a liveness property of a program. In our scenario, the liveness property means that, when an event occurs, the event builder computation reacts to the event and eventually reaches another waiting point.

We believe that the structuring of reactive programs presented in this thesis would make the automatic verification easier compared to, for example, standard programming model that builds heavily on mutable state. In our programming model, more information about the program is encoded directly in the structure of the code and the tools for automated verification of program could likely benefit from this information. Evolving our programming model is an interesting direction as it would enable using it in mission critical scenarios.

5. Real-world reactive scenarios

Another future direction for the work presented in this thesis is to evaluate our reactive programming library in several real-world scenarios. So far, we mainly focused on programming of reactive user-interfaces of windows applications, which is a very frequent problem. However, there are several other, very interesting, areas for evaluating reactive programming models.

Programming robots. The Functional reactive framework Yampa [63] has been used for programming robots and uses many appealing examples from this field. Robots are, indeed, a very reactive system – most of the sensors provide the input in form of events and the robot needs to (quickly) respond to these events. Moreover, we often need to react to various combinations of events and our programming model appears to be very suitable in this case.

Technically speaking, using our F# library for programming robots may be very well possible using the Microsoft Robotics Studio[10] tools. The programming model in Robotics Studio is extremely complex compared to our approach, so the use of a lightweight model presented in this thesis would be a large simplification. However, a question is how difficult would it be to build our library on top of the programming model available in Robotics Studio.

Client-side web programming. Another problem domain where most of the applications are extremely reactive is the client-side of a web application. A modern web application needs to react to events caused by the user (similarly to a desktop application), but in addition, it also needs to communicate with the server, which is done via asynchronous invocations. The client receives a response from the server using an event that it needs to handle.

Technically, there are three ways for using our programming model on the client-side of a web application. A somewhat limited approach is to implement the programming model as a JavaScript library. The viability of this approach has been demonstrated in Chapter III, however the JavaScript language doesn't support any equivalent to F# asynchronous workflows, so the encoding would be cumbersome.

[10] More information about MS Robotics Studio is available at: http://msdn.microsoft.com/robotics

Another option is to use the Silverlight platform, which allows us to run .NET applications on the client-side using a web browser extension. This approach is appealing as we could write application using the F# language and use the library "as it is". Although the Silverlight platform isn't as wide-spread as JavaScript, there is a relatively large number of applications developed using Silverlight, so it would not be difficult to find users for our reactive library.

Finally, the last option is to use the F# language together with a translator (or a compiler) that produces JavaScript code. We demonstrated that this approach is perfectly possible in the Bachelor thesis of the author [67]. This would give us all the benefits of the F# language, but we would not rely on any specific programming environment. The developed reactive programs would only require support for JavaScript, which is assumed when creating rich web applications.

Chapter VIII

Overview of contributions and conclusion

The main goal of this thesis was the development of a reactive library that is based on the imperative programming model and can be integrated with the existing, declarative combinators for working with events. Our aim was to develop a library that is clearly specified, so that it is possible to formally reason about the programs written using the library.

Aside from the reactive library itself (presented in Chapter VI), we presented several results that are interesting on their own and could be used and further developed separately from our reactive programming model. In this chapter, we first review the results presented in this thesis that we find the most interesting and then we briefly conclude the thesis.

1. Overview of contributions

1.1 Garbage collection

Our first contribution is in the field of garbage collection. We formally described the problem of garbage collection in the reactive programming scenario based on events. We provided a simple and arguably elegant definition of collectability for events based on the duality principle. To our knowledge, we are the first to use the duality principle for discussing the memory model of reactive applications.

Next, we combined the collectability of events with the usual collectability of objects to provide a definition that is suitable for the usual mixed scenario. Using this definition, we presented a garbage collection algorithm, which reclaims all collectable objects and events. The algorithm is based on graph transformation, which allows us to reduce the problem to well-known GC algorithms. We also presented a proof of the algorithm correcntess.

However, our implementation aim wasn't to actually replace a garbage collection algorithm. Instead, we have shown an alternative implementation of library of F# event combinators, solely in terms of object references. Our implementation closely corresponds to the formal model and doesn't cause memory leaks when used in an environment that combines both declarative and imperative approach to reactive programming.

1.2 Pattern matching for monads

The key result presented in Chapter V is that a wide range of reactive and concurrent programming models can be encoded using a simple and reusable language extension, without the need to design a specialized language for each programming model. We presented a language feature that extends F# computation expressions with monadic pattern matching and we sketched numerous applications ranging from lists processing to concurrent programming.

Our language extension is based on pattern matching construct known from many functional programming languages. We integrated it in the F# language, which is based on ML, so we made a special effort to preserve the user's existing intuition about pattern matching. By requiring several simple laws about basic combinators, we can guarantee numerous results that are helpful for reasoning about our monadic pattern matching.

Aside from practical applications, our work is also shows an interesting relation between monadic pattern matching and commutative monads. In particular, our construct can be used for binding on multiple monadic values in parallel using a syntax that is less sequential than the one used by standard monads.

1.3 Semi-discrete time in reactive applications

The widely used approaches for reasoning about reactive applications use either discrete time or continuous time. In the first case, a program consists of a series of steps. An event occurs repeatedly at certain steps and a reaction to an event is performed in a single step. In the second case, an event is associated with a precise time value (modelled for example as a floating point number). In principle, this means that another event may occur between each two event occurrences.

We develop a novel way of modelling time in reactive applications. It allows us to represent the fact that a reaction to an event may trigger multiple events (that occur at distinguishable times) in response, but that all occur before the original event may be triggered again. This makes it possible to reason about our imperative programming model formally.

1.4 Imperative monad-like computation

Finally, our last contribution is an abstract algebraic description of imperative computations that are based on the same primitives as monads, but obey a different set of laws. When we compose the *bind* operation with the *return* operation of a standard monad, we get an identity function. However, our implementation of *bind* is imperative and waits only for the first occurrence of an event and as a result it doesn't obey this law. However, it is still possible to implement an identity function using recursion.

We believe that these types of computations may appear more often in impure functional languages that support monads and we demonstrate this claim by presenting a computation for working with an imperative representation of a sequence (in addition to our computation for working with events). We also formulate an alternative algebraic law that describes the implementation of identity function on our imperative computations using recursion.

2. Conclusion

We believe that reactive applications will become the most important type of programs over the next few years. The need for richer and more interactive user interfaces, larger emphasis on distributed systems as well as modern trends like programming for the cloud all contribute to this direction.

Even though there are several approaches known to the research community, the solutions usually used in practice are surprisingly hard to use and, more importantly are very difficult to use correctly. The reactive programming scenario brings many challenges that complicate the life of programmers. We need to ensure that programs correctly respond to all situations, that they don't get stuck on some rare occasions and that concurrent events are handled carefully.

The approach presented in this thesis is both theoretically well founded and practical and easy to use. The way our programs are structured makes it easy to encode many frequent programming patterns that appear in reactive programming. It makes it possible to test programs using unit-testing techniques and it provides better abstractions for developing reusable components. Moreover, our programming model is easier to reason about, both formally and informally.

This thesis presents several novel ideas that contribute to the development of better reactive programming models. Our work already contributed to both academic and programming community. Some of the work presented in this thesis has been published in refereed literature and some of the techniques that we develop are now being actively used for real-world F# projects.

References

[1] C. Elliott and P. Hudak, Functional reactive animation. *In Proceedings of ICFP 1997, pp. 263-273*

[2] C. Elliott. Declarative event-oriented programming.
In Proceedings of PPDP 2000

[3] E. Scholz. Imperative streams - a monadic combinator library for synchronous programming. *In Proceedings of ICFP 1998*

[4] D. Syme, A. Granicz, and A. Cisternino. Expert F#, Reactive, Asynchronous and Concurrent Programming. *Apress, 2007.*

[5] E. Meijer. LiveLabs Reactive Framework. *Lang.NET Symposium 2009, Available at: http://tinyurl.com/llreactive*

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, R. De Simone. The Synchronous Languages Twelve Years Later. *In Proceedings of the IEEE, vol. 91, pp. 64-83, 2003*

[7] E. Meijer, B. Beckman, G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework.
In Proceedings of COMAD 2006

[8] T. Petricek, D. Syme. Collectable event chains in F#.
To appear as MSR Technical Report.

[9] Microsoft. Reactive Extensions for .NET. *Retrieved from: http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx,2010*

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, LUSTRE: A declarative language for programming
synchronous systems. *In Proceedings of POPL 1987.*

[11] F. Boussinot and R. de Simone, The Esterel language. *In proceedings of the IEEE, vol. 79, pp. 1293–1304, 1991.*

[12] G. Berry and G. Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *In Science of Computer Programming vol. 19, n°2, pp 87-152, 1992.*

[13] J. Armstrong, R. Virding, C. Wikström and M. Williams, Concurrent Programming in ERLANG, 2nd ed. *Prentice Hall International Ltd., 1996.*

[14] D. Kafura, D. Washabaugh, and J. Nelson, Garbage collection of actors. *In OOPSLA'90, vol. 25(10), pp. 126–134*

[15] A. Vardhan and G. Agha, Using Passive Object Garbage Collection Algorithms for Garbage Collection of Active Objects. *In Proceedings of ISMM'02*

[16] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. *MIT Press, Cambridge, Mass., 1986.*

[17] Z. Wan and P. Hudak. Functional Reactive Programming from First Principles. *In Proceedings of PLDI, 2000*

[18] T. Petricek and J. Skeet. Real-World Functional Programming, Chapter 16, *Manning, 2010*.

[19] D. Syme. Simplicity and Compositionality in Asynchronous Programming through First Class Events. *Online at: http://tinyurl.com/composingevents, Retrieved: Jan 2010*

[20] D. Syme. Initializing Mutually Referential Abstract Objects. *In Proceedings of ML Workshop, 2005*

[21] G. Schechter. Simulating "Weak Delegates" in the CLR. *Online at: http://tinyurl.com/weakdelegates, Retrieved: Feb 2010*

[22] O. Agesen, D. Detlefs and J. Eliot B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In Proceedings of PLDI 1998.

[23] R. Shaham, E. K. Kolodner and M. Sagiv. Estimating the Impact of Heap Liveness Information on Space Consumption in Java. In Proceedings of ISMM 2002.

[24] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In OOPSLA 2009.

[25] The JQuery Project. jQuery. Available at http://jquery.com

[26] B. Hayes. Ephemerons: a new finalization mechanism.
In Proceedings of OOPSLA 1997.

[27] T. Petricek, D.Syme. Collecting Hollywood's Garbage: Avoiding Space-Leaks in Composite Events (Extended version). Available at:
http://tomasp.net/academic/event-chains.aspx

[28] D. Syme, A. Granicz, and A. Cisternino. Expert F#, Introducing Language-oriented Programming. *Apress, 2007.*

[29] P. Wadler. Monads for functional programming. In Advanced Functional Programming, LNCS 925, 1995.

[30] Microsoft. F# Language Specification. Available online at:
http://tinyurl.com/fsspec, Retrieved February 2010

[31] C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In Proc. POPL 1996.

[32] C. Fournet,, F. Le Fessant, L. Maranget, A Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In Advanced Functional Programming, vol. 2638 of LNCS, pp 129–158. Springer, 2002.

[33] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. ACM Trans. Program. Lang. Syst, 26(5):769–804, 2004.

[34] C. Russo. The Joins concurrency library. In PADL 2007.

[35] S. Singh. Higher-order combinators for join patterns using STM. In Proc. TRANSACT Workshop, OOPSLA, 2006.

[36] P. Haller, T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. In Proc. COORDINATION 2008.

[37] H. Baker, C. Hewitt. "The Incremental Garbage Collection of Processes". In Proc. Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12.

[38] M. Fluet, M. Rainey, J. Reppy and A. Shaw. Implicitly-threaded parallelism in Manticore. In Proc. ICFP 2008

[39] Syme, D., G. Neverov, J. Margetson. Extensible Pattern Matching via a Lightweight Language Extension. ICFP, 2007.

[40] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Proc. POPL, 1987

[41] S. P. Jones. Wearing the hair shirt - A retrospective on Haskell. Invited talk, POPL 2003. Slides available online at: http://tinyurl.com/haskellretro

[42] McBride, C. and R. Paterson, Applicative programming with effects, Journal of Functional Programming 18 (2008)

[43] S. P. Jones (ed.) Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, 2003.

[44] M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller. Data Parallel Haskell: a status report. In Proc. Workshop on Declarative Aspects of Multicore Programming, 2007.

[45] D. King, P. Wadler. Combining Monads. In Proc. of Glasgow Workshop on Functional Programming, 1992.

[46] C. Okasaki. Views for Standard ML. In Proc. Workshop on ML, Baltimore, Maryland, USA, pp. 14–23, 1998.

[47] J. Hughes, Generalising Monads to Arrows, in Science of Computer Programming 37, pp67-111, May 2000.

[48] R. Paterson. A new notation for arrows. In ICFP 2001

[49] Hai Liu. E. Cheng. P. Hudak. Causal Commutative Arrows and Their Optimization. In Proc. ICFP 2009

[50] B. Emir, Odersky, M., Williams, J. Matching Objects with Patterns. *In Proceedings of ECOOP 2007.*

[51] Ma Qin, L. Maranget. Compiling Pattern-Matching in Join-Patterns, In Proc. CONCUR 2004

[52] R. Kieburtz. Codata and Comonads in Haskell. Unpublished draft, 1999. http://tinyurl.com/comonads

[53] T. Uustalu, V. Vene. The essence of dataflow programming. In Proceedings of APLAS 2005

[54] R. Milner, M. Tofte, R. Harper, D. MacQueen. The Definition of Standard ML – Revised, ISBN: 978-0262631815, The MIT Press, 1997

[55] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software (Iterator pattern, Chapter 5), ISBN: 978-0201633610, Addison-Wesley Professional, 1994

[56] P. Hudak. The Haskell School of Expression: Learning functional programming through multimedia. ISBN: 978-0521644082, Cambridge University Press, 2000

[57] J. H. Reppy. Concurrent Programming in ML. ISBN: 978-0521714723, Cambridge University Press, 2007

[58] D. Syme. Leveraging .NET meta-programming components from F#: Integrated queries and interoperable heterogeneous execution. *In Proceedings of Workshop on ML and its Applications, 2006.*

[59] E. Moggi. Notions of computation and monads.
In Information and Computation, 93:55-92, 1991.

[60] P. Wadler. Comprehending monads.
In Mathematical Structures in Computer Science, 1992, pp. 61-78.

[61] J. C. Mitchell. Concepts in programming languages. ISBN: 978-0521780988, Cambridge University Press, 2001

[62] T. Petricek with J. Skeet. Real-World Functional Programming With examples in F# and C#. ISBN: 978-1933988924. Manning, 2009.

[63] P. Hudak, A. Courtney, H. Nilsson, J. Peterson. Arrows, robots, and functional reactive programming. In Advanced Functional Programming, 4th International School 2002, LNCS vol. 2638, pp. 159–187. Springer-Verlag, 2003.

[64] M. Jauernig. Rx: Event composition – multi-valued. Unpublished, Available at: http://www.minddriven.de/?p=563

[65] M. Tullsen. First class patterns. In Proceedings of PADL, 2000

[66] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In Proceedings of POPL, 2007

[67] T. Petricek. Client side scripting using meta-programming. Bachelor thesis, Charles University in Prague, 2007. Available at: http://tinyurl.com/fswebtools

Appendix A

Imperative and object-oriented F#

The authors of the F# language give the following description in [58]: "F# is a multi-paradigm .NET language explicitly designed to be an ML suited to the .NET environment. It is rooted in the core ML design and in particular has a core language largely compatible with OCaml." A more formal specification of the language is available in [30].

In the above description, the term multi-paradigm refers to the fact that F# is primarily a functional programming language; however, it is not a pure language and provides numerous imperative features that make the language easily interoperable. F# also provides a full support for the .NET object model and can be used for developing object-oriented as well as functional (compositional) libraries. We already demonstrate some of the functional features of F# in Chapter II, so this appendix focuses only on imperative and object-oriented aspects.

3. Imperative programming

By default, all types and values declared in F# are immutable, which means that they cannot be modified once they are created. However, as F# is a .NET language, we sometimes need to use mutation in order to integrate F# code with existing .NET libraries that rely on mutation. For this reason, F# provides many imperative constructs. However, their use is generally discouraged.

Using mutable values. At the basic level, we can specify that a value of a variable declared using `let` binding can be changed by adding the `mutable` modifier. This may break the referential transparency of the program, but it is sometimes needed. The following example shows an imperative implementation of the well known factorial function:

```
1: let factorial x =
2:   let mutable res = 1
3:   for i in 1 .. x do
4:     res <- res * i
5:   res
```

The code first declares a mutable variable (line 2) and then uses a standard imperative looping construct and assignment (line 4) which is written as `<-` in F#. Note that the `factorial` function is still purely functional if we look at its behavior and

not at its implementation. The function always returns the same result for a given argument, which is a defining property of pure functions.

Working with mutable types. More frequent case where we use mutation in F# is when writing code that uses some .NET libraries as most of the standard .NET types are designed as mutable. The following example demonstrates working with the `ResizeArray<'a>` type, which implements a mutable collection. After creating an instance of the collection, we use the `Add` for adding new elements:

```
1: let list = new ResizeArray<_>()
2: list.Add("Ahoj")
3: list.Add("Bonjour")
4: list.Add("Hello")
5:
6: for s in list do
7:   Console.WriteLine(s)
```

The snippet starts by creating an instance of the mutable collection (line 1). Then we add three elements to the collection using the `Add` method (lines 2, 3 and 4) and finally, we print all the elements using a `for` loop (lines 6, 7). It is also worth mentioning that we didn't specify the type of the elements stored in the collection when creating it (line 1) and we simply used the underscore symbol. This instructs the compiler to infer the type automatically. In our example, the compiler sees that we're adding elements of type `string`, so it deduces that the type of the created collection has to be `ResizeArray<string>`. The F# language is a statically typed language equipped with type inference, which means that if we attempted to add an element of another type (e.g. a number) we would get a compile-time error.

4. Object-oriented programming

In this section we'll review object-oriented features of F# that we'll use in some of the examples presented later in this thesis. The F# language uses the .NET object model, which means that it supports a combination of single class inheritance with a multiple interface inheritance. The use of class inheritance in F# is relatively rare, so we discuss only working with interfaces and declaration of simple classes.

Declaring interfaces. The F# language attempts to unify functional and object-oriented type declarations, so the syntax for declaring an interface is similar to other type declarations. Moreover, when declaring object-oriented types, F# automatically infers what kind of type we are declaring. When the type doesn't contain a constructor and consist only of abstract members, then it is considered as an interface, when the type contains a constructor or an instance method, it is considered as an abstract class and so on (however, it is possible to specify the kind of type explicitly if we need to). The following declaration creates a simple interface with a single method that takes string as an argument and returns an integer:

```
type FeatureExtractor =
  abstract Extract : string -> int
```

The type represents an abstract feature extractor that returns some information about the given string (and could be for example used to test whether a password is safe or in some machine learning scenario).

Using object expressions. The easiest way to obtain an implementation of interface is to use object expression. Object expressions correspond to anonymous classes in Java and allow us to construct an implementation of an interface "on the fly" without declaring a new type. The following example creates an implementation of `FeatureExtractor` that counts the number of non-letter characters in the string:

```
1: let letters =
2:   { new FeatureExtractor with
3:       member x.Extract(s) =
4:         s |> Seq.filter (not << Char.IsLetter) |> Seq.length }
5:
6: > letters.Extract("hello world!!!");;
7: val it : int = 4
```

The code declares a value named letters and initializes it with an implementation of the `FeatureExtractor` interface created using object expression (lines 2-4). The object expression contains methods required by the interface that are written using the `member` keyword. The implementation of the `Extract` method (line 4) uses higher-order functions (we're working with general sequences from the `Seq` module rather than functions for working with functional lists from `List` module). We first filter all characters from the string that are not a letter. To do this, we compose the `Char.IsLetter` function with a function `not` using the *function composition* operator written as `<<`. Next, we count the number of characters in the resulting sequence. The example on the last two lines of the listing shows that a sample string contains 4 non-letter symbols.

Declaring classes. In our last example, we'll create a simple class type with a constructor and several public members. Just like in the interface declaration, we'll use the `type` keyword to do this. The difference is that we'll use *implicit constructor* declaration (by providing an argument list immediately after the name of the type) and we'll add several properties and a method to the type using the `member` keyword. The class provides simple functionality for working with passwords:

```
1: type Password(str:string) =
2:   let count f =
3:     str |> Seq.filter f |> Seq.length
4:   member x.Digits = count Char.IsDigit
5:   member x.NonLetter = count (not << Char.IsLetterOrDigit)
6:   member x.Contains(sub) = str.Contains(sub)
```

The implicit constructor (line 1) takes a single parameter named `str`. We need to explicitly specify the type of the parameter in this case as we later invoke methods on the `str` value and the F# compiler doesn't have any cues on what the type of the value is. The parameters of the constructor are automatically accessible in all the members of the type (under the cover, they are stored in a field if they are used from any of the members).

Next, we declare a local helper function `count`, which is private to the type and counts the number of characters matching the specified predicate. The function is used by the first two members (lines 4 and 5) that return the number of characters that are digits and the number of non-alphanumerical characters, respectively. Finally, the last member of the type is a method named `Contains` which tests whether the string contains some specified substring.

Appendix B

Counting clicks using combinators

In section 2.3, we used imperative reactive programming techniques to implement a click counter which limits the rate of clicks to at most once click per second. This means that when the user clicks on a button, all clicks will be ignored for the next one second. We used the `Async.Sleep` function to pause the asynchronous workflow, which implemented the behavior. We claimed that the when implementing the same behavior using F# event combinators, the code becomes less readable. To illustrate the point we will now show one possible declarative implementation. Note that the following example isn't purely functional, because it uses global mutable value `DateTime.Now`[11]:

```
1: let clickCounter =
2:   btn.MouseDown
3:     |> Event.map (fun _ -> DateTime.Now)
4:     |> Event.scan (fun (_, dt:DateTime) ndt ->
5:          if ((ndt - dt).TotalSeconds > 1.0) then
6:            (1, ndt) else (0, dt))
7:                   (0, DateTime.Now)
8:     |> Event.map fst
9:     |> Event.scan (+) 0
```

Whenever the source event occurs, we take the current time (line 3). The stateful `scan` combinator (line 4) remembers the last time event has occurred. It checks whether the delay was long enough (line 5). If yes, it yields 1 and remembers the current time otherwise it yields 0 without updating the last occurrence time.

Next, the processing pipeline drops the time from the tuple yielded by `Event.scan` (line 8). The result carries 0 each time the button click was ignored or 1 when the interval between clicks is long enough. Finally, we use `Event.scan` again to count the total number of clicks (line 9).

[11] To make the code pure, we'd have to represent the current time as an event. However, this isn't possible in F# and it would require more sophisticated declarative reactive programming model (e.g. as in [17]).

Appendix C

Merge for commutative monads

In section 4.2 of Chapter V, we discussed an interesting relation between monads that provide the *merge* operation (obeying the laws discussed in section 3.1 of Chapter II) and commutative monads (obeying the usual monad laws and an additional commutativity law). In particular, we stated that, for commutative monads, it is possible to implement ⫫ in terms of *bind* and *return*.

In this appendix, we present a proof that the implementation (shown in Chapter V) obeys the required laws. The proof is relatively straightforward, but is shown here for completeness. We use the following specialization of the associativity law (a may appear as a free variable in *expr*):

$$
\begin{aligned}
& (m \gg\!= (\backslash a \to expr)) \gg\!= g \\
= \quad & (associativity) \\
& m \gg\!= (\backslash a \to ((\backslash a \to expr)\ a) \gg\!= g) \\
= \quad & (\beta\ reduction) \\
& m \gg\!= (\backslash a \to expr \gg\!= g)
\end{aligned}
$$

4.1 (C1) Merging two returns produces a tuple

$$
\begin{aligned}
& (return\ a) \fatsemi (return\ b) \\
= \quad & (definition\ of \fatsemi) \\
& (return\ a) \gg\!= (\backslash a \to return\ b \gg\!= (\backslash b \to return\ (a, b))) \\
= \quad & (left\ identity) \\
& (return\ a) \gg\!= (\backslash a \to return\ (a, b)) \\
= \quad & (left\ identity) \\
& return\ (a, b)
\end{aligned}
$$

4.2 (C2) Associativity

$$
\begin{aligned}
& map\ assoc\ ((a \fatsemi b) \fatsemi c) \\
= \quad & (definition\ of\ map) \\
& ((a \fatsemi b) \fatsemi c) \gg\!= (return \cdot assoc) \\
= \quad & (definition\ of \fatsemi) \\
& ((a \fatsemi b) \gg\!= (\backslash x \to c \gg\!= (\backslash c \to return\ (x, c)))) \gg\!= (return \cdot assoc) \\
= \quad & (definition\ of \fatsemi)
\end{aligned}
$$

$$(a \gg= (\backslash a \to b \gg= (\backslash b \to return\ (a,b)))) \gg= (\backslash x \to c \gg= (\backslash c \to return\ (x,c))) \gg= (return \cdot assoc)$$

$= \quad (associativity, 3x)$

$$(a \gg= (\backslash a \to b \gg= (\backslash b \to return\ (a,b) \gg= (\backslash x \to c \gg= (\backslash c \to return\ (x,c)))))) \gg= (return \cdot assoc)$$

$= \quad (left\ identity)$

$$(a \gg= (\backslash a \to b \gg= (\backslash b \to c \gg= (\backslash c \to return\ ((a,b),c))))) \gg= (return \cdot assoc)$$

$= \quad (associativity, 4x)$

$$a \gg= (\backslash a \to b \gg= (\backslash b \to c \gg= (\backslash c \to return\ ((a,b),c) \gg= (return \cdot assoc))))$$

$= \quad (left\ identity)$

$$a \gg= (\backslash a \to b \gg= (\backslash b \to c \gg= (\backslash c \to return\ (assoc\ ((a,b),c)))))$$

$= \quad (definition\ of\ assoc)$

$$a \gg= (\backslash a \to b \gg= (\backslash b \to c \gg= (\backslash c \to return\ (a,(b,c)))))$$

$= \quad (left\ identity;\ backwards)$

$$a \gg= (\backslash a \to b \gg= (\backslash b \to c \gg= (\backslash c \to return\ (b,c) \gg= (\backslash x \to return\ (a,x)))))$$

$= \quad (associativity;\ 2x;\ backwards)$

$$a \gg= (\backslash a \to b \gg= (\backslash b \to c \gg= (\backslash c \to return\ (b,c))) \gg= (\backslash x \to return\ (a,x)))$$

$= \quad (definition\ of\ \textcircled{\scriptsize ||};\ backwards)$

$$a\ \textcircled{\scriptsize ||}\ (b \gg= (\backslash b \to c \gg= (\backslash c \to return\ (b,c))))$$

$= \quad (definition\ of\ \textcircled{\scriptsize ||};\ backwards)$

$$a\ \textcircled{\scriptsize ||}\ (b\ \textcircled{\scriptsize ||}\ c)$$

4.3 (C3) Commutativity using laws of commutative monads

$$map\ swap\ (b\ \textcircled{\scriptsize ||}\ a)$$

$= \quad (definition\ of\ map)$

$$(b\ \textcircled{\scriptsize ||}\ a) \gg= (return \cdot swap)$$

$= \quad (definition\ of\ \textcircled{\scriptsize ||})$

$$(b \gg= (\backslash b \to a \gg= (\backslash a \to return\ (b,a)))) \gg= (return \cdot swap)$$

$= \quad (associativity)$

$$b \gg= (\backslash b \to (a \gg= (\backslash a.return\ (b,a))) \gg= (return \cdot swap))$$

$= \quad (associativity)$

$$b \gg= (\backslash b \to a \gg= (\backslash a \to return\ (b,a) \gg= (return \cdot swap)))$$

$= \quad (left\ identity)$

$$b \gg= (\backslash b \to a \gg= (\backslash a \to return\ (swap\ (b,a))))$$

$= \quad (definition\ of\ swap)$

$$b \gg= (\backslash b \to a \gg= (\backslash a \to return\ (a,b)))$$

$= \quad (commutativity)$

$$a \gg= (\backslash a \to b \gg= (\backslash b \to return\ (a,b)))$$

$= \quad (definition\ of\ \textcircled{\scriptsize ||};\ backwards)$

$$a\ \textcircled{\scriptsize ||}\ b$$