

# Pattern matching for monadic computations

(Unpublished draft)

Tomáš Petříček

Faculty of Mathematics and Physics  
Charles University in Prague, Czech Republic  
tomas@tomasp.net

Don Syme

Microsoft Research, Cambridge, UK  
dsyme@microsoft.com

## Abstract.

Modern programming challenges led to a design of programming models for specific domains, such as reactive, parallel and concurrent programming. Unfortunately, these programming models are often difficult to encode in a general purpose language.

We present an extension of monadic computations that adds support for pattern matching on monadic values. This allows us to embed many of the useful programming models in a general purpose language with an easy to use syntax. Moreover, our extension keeps not only a familiar syntax, but also a familiar semantics of pattern matching, which makes it easy to reason about it, even in a non-standard programming model.

Using this extension, we can define for example computations for reactive programming, concurrent programming model based on join calculus and parallel programming using futures. All these three encodings are implemented as a library extension that benefits from the syntax we introduce. Moreover, all of them follow similar principle, which makes them very easy to use.

**Categories and Subject Descriptors** D.3.2 [Programming Languages]: Language Classifications – applicative (functional) languages; concurrent, distributed and parallel languages; data-flow languages

**General Terms** Design, Languages

## 1. Introduction

Today, we often need to write programs for an environment that is in some way non-standard. In parallel programming, we can execute multiple functions at one time; in concurrent programming, we need to synchronize executing processes that can communicate; in reactive programming, we write code that waits for one or more events (such as user input or a server response) and acts in response.

One way to solve these challenges is to design a specialized language, which provides a clear solution with a comfortable syntax, but it also binds the language to a single programming model. On the other hand, when developing the solution solely as a library, we have a full freedom about the programming model, but we can use only a restricted syntax, which lowers the usability of the solution. An approach that lies in between is to identify a repeating pattern in the library-based solutions and to provide a

syntactic sugar that can be used by a large number of libraries.

This approach is successfully utilized by Haskell's monads [2] as well as some more recent extensions (e.g. arrows [22] and applicative functors [16]) and is also used by computation expressions in F# [1]. 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 areas such as reactive, parallel and concurrent programming.

In this paper, we show how to extend monadic computations with pattern matching on monadic values and we look at several applications of this feature. The key contributions of this paper are the following:

- Most importantly, 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 3 supports this claim by example. We show a reactive programming model (section 3.1) inspired by [17, 23]; a concurrent programming model (section 3.3) based on join calculus [5] bearing similarities to [6, 7]; and a parallel programming model (section 3.4) based on futures, which can express some features of Manticore [12].

- 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 4).

We define laws that should hold about the two operations (Sections 5 and 6). We observe interesting relationship with commutative monads and show that our syntactic extension is useful when working with them (Section 5.2).

- 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 7).

We conclude this paper by discussing several design alternatives that may be relevant for other languages or other computation types (Section 8). Our work is built on top of F# computation expressions which we introduce in the next section. However, the presented ideas should be applicable to any language with syntactic support for monads.

## 2. Motivation

Before we look at several examples that motivate the work presented in this paper, we'll briefly introduce *computation expressions*, an F# language feature that has been largely inspired

<code>expr</code>	<code>= expr { cexpr }</code>	Computation expression
<code>cexpr</code>	<code>= let pat = expr in cexpr</code>	Binding value
	<code>  let! pat = expr in cexp</code>	Binding computation
	<code>  return expr</code>	Return result
	<code>  return! expr</code>	Return computation
	<code>  match expr-list with</code> <code>pat-list<sub>i</sub> -&gt; cexpr<sub>i</sub></code>	Value pattern matching

## 2.2 Reactive programming

In our first example, we look at writing event-driven applications. We can think of an event as a sequence of pairs containing a time and a value. They can be constructed using combinators such as

	$\llbracket - \rrbracket_{\text{cexpr}} : \text{cexpr} \times \text{ident} \rightarrow \text{expr}$
	$\llbracket \text{expr } \{ \text{cexpr} \} \rrbracket_{\text{cexpr}} = \text{let } m = \text{expr in } \llbracket \text{cexpr} \rrbracket_{\text{cexpr}} m$
	$\llbracket \text{let pat = expr in cexpr} \rrbracket_{\text{cexpr}} m = \text{let pat = expr in } \llbracket \text{cexpr} \rrbracket_{\text{cexpr}} m$
	$\llbracket \text{let! pat = expr in cexpr} \rrbracket_{\text{cexpr}} m = \text{bind}_m (\text{fun pat} \rightarrow \llbracket \text{cexpr} \rrbracket_{\text{cexpr}} m) \text{ expr}$
	$\llbracket \text{return expr} \rrbracket_{\text{cexpr}} m = \text{return}_m \text{ expr}$
	$\llbracket \text{return! expr} \rrbracket_{\text{cexpr}} m = \text{expr}$
	$\llbracket \text{match expr-list with} \quad \text{match expr-list with} \rrbracket_{\text{cexpr}} m = \text{pat-list}_i \rightarrow \llbracket \text{cexpr}_i \rrbracket_{\text{cexpr}} m$

Figure 1. Syntax and translation rules for F# computation expressions

by Haskell monads. The similarity is apparent, so our work would be also directly applicable to Haskell.

### 2.1 Computation expressions

Computation expressions are based on monads. As noted in [1], they are named differently as they usually serve somewhat different purpose. They are defined mainly in terms of two well-known combinators that are used to compose computations. The type of the operators for some monad  $M$  is:

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

Similarly to Haskell monads, the two combinators should also satisfy the usual *monad laws* as described in [2]. F# combines the two operations into a *computation builder*, which is used to denote the monadic type of a computation. The most prominent example in F# is `async`, which defines an asynchronous workflow [4]. It is essentially a one-shot continuation monad and can be used for executing long-running computations without blocking threads:

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

The function downloads a web page. Getting the initial response from the server and downloading the content may take a long time, so these operations are executed asynchronously using `bind`.

We include a minimal formal specification of F# computation expressions suitable for the examples in this paper. The Figure 1 shows the core syntax and translation rules that define it in terms of `bind` and `return`. The rules do not include exception handling, looping constructs and support for writing monoids. More information is available in the F# specification [3]. The F# type system doesn't support higher kind 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.

An equivalent definition of monads uses three operations: `join`, `map` and `unit`. These can be implemented as follows:

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

Indeed, it is also possible to define `bind` and `return` in terms of the other three operations. We find it clearer to use `map` and `join` in some examples, so we'll often combine the definitions.

`Event.filter` (which turns an existing event into a new event that is triggered only when the value carried as an argument matches the specified predicate). However, we can also work with events using computation expressions.

We embed events into asynchronous workflows by writing an impure function that creates a workflow, which waits for the first occurrence of an event and returns the value carried by the event. This way, we can express many processing patterns as recursive computations that await the first occurrence, process the value and then recursively call themselves to wait for the next value. We look at a function for drawing a rectangle. After pushing the button, we enter a loop in which we need process mouse move (to update the rectangle) and release of the button (to finish drawing):

```
1: let rec drawing() = event {
2:   let! e = await (w.MouseMove <|> w.MouseUp)
3:   match e with
4:   | Choice1Of2(m) -> redraw (m.X, m.Y)
5:   | Choice2Of2(m) -> 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 `<|>` that creates an event which is triggered whenever any of the argument triggers. The combined event is passed to the `await` 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). Our programming model serializes events, so no events can happen while processing a previous occurrence.

The important point is that we need to leave the computation expression sub-language 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 `<|>`.

### 2.3 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: > calcDiffs [5; 5; 1] [5; 4; 8];;
7: 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).

`List.zip` is in some sense similar to the previous `<|>` combinator. Both of them allow us to combine two monadic values (`List<'a>` or `Event<'a>` respectively) into a single one. The difference is that `<|>` gives us only a single unwrapped value at the time (from one of the events) and `List.zip` gives us both values as a tuple. In a way, `<|>` 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 (we require both values).

### 2.4 Concurrent programming

Our last motivation is based on Join calculus [5], which provides a declarative way for expressing synchronization patterns. Joins have been used as a basis for language features [6, 7], but it is also possible to implement them as a library [8, 9]. Scala provides an elegant integration thanks to extensible pattern matching [10].

The following example shows a simple unbounded FIFO buffer implemented in Scala. It uses two channels and consists of just a single join pattern:

```
1: val Put = new AsyncEvent[Int]
2: val Get = new SyncEvent[Int]
3: join { case Get() & Put(num) =>
4:   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 3). It specifies that the body (line 4) 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) }
```

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.

## 3. 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.

### 3.1 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'll require the following *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 3 and focus on an example for now.

When merging two lists in section 2.3, 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.

### 3.2 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 2.2, 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<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 2.2, 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() = event {
2:   match! await w.MouseMove, await w.MouseUp with
3:   | !m, _ -> redraw (m.X, m.Y)
4:   | _      -> return! drawing()
5:   | _, !m -> return m.X, m.Y }
```

As already explained in section 3.1, 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.

### 3.3 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 2.4, 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:   | !chn1, !n, _ ->
8:     chn1.Reply("Number: " + n.ToString())
9:   | !chn1, _, !s ->
10:    chn1.Reply("String: " + s) }
```

Each clause combines two channels (lines 7 and 9) and ignores the third one. If we get an integer value and a reply channel `chn1` in the first join pattern (line 7), we send a number converted to a string as the reply (line 8). The second join is quite similar.

### 3.4 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 [11]). 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 create 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 5. 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

<code>cpat</code>	<code>= _</code>	Ignore pattern
	<code>  !pat</code>	Binding pattern
<code>ccl</code>	<code>= cpat<sub>1</sub>, ..., cpat<sub>k</sub> -&gt; cexpr</code>	Computation match clause
	<code>  cpat<sub>1</sub>, ..., cpat<sub>k</sub> when</code>	Computation match clause
	<code>expr -&gt; cexpr</code>	...with guard condition
<code>cexpr</code>	<code>= match! expr<sub>1</sub>, ..., expr<sub>k</sub> with</code>	Monadic pattern matching
	<code>ccl<sub>1</sub>   ... ccl<sub>l</sub></code>	...sequence of clauses
	<code>  ...</code>	(other computations)

Figure 2. Syntax of monadic pattern matching

`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 [12]. The parallelism in Manticore is implicit. However, we achieve similar syntactic simplicity and expressive power just by using a generally useful language feature.

## 4. 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 [1] and active patterns [13] 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.

### 4.1 Syntax

The syntax of our extension is shown in Figure 2. In addition the standard constructs described in Figure 1, 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 [14] 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 [13] 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`.

## 4.2 Translation

In this section, we describe a translation that transforms pattern

$$\llbracket \text{expr } \{ \text{cexpr} \} \rrbracket = \text{let } m = \text{expr} \text{ in } \llbracket \text{cexpr} \rrbracket_{\text{cexpr } m} \quad (1)$$

$$\llbracket \text{cpat}_1, \dots, \text{cpat}_k \rightarrow \text{cexpr} \rrbracket_{\text{ccl } m, (v_1, \dots, v_k)} = \text{map}_m (\text{function } | (\text{pat}_1, \dots), \text{pat}_n \rightarrow \text{Success}(\text{fun } () \rightarrow \llbracket \text{cexpr} \rrbracket_{\text{cexpr } m}) | \_ \rightarrow \text{Failure}) \text{cargs} \quad (2)$$

$$\text{where } \{ (\text{pat}_1, v_1), \dots, (\text{pat}_n, v_n) \} = \{ (\text{pat}_i, v_i) \mid \text{cpat}_i = !\text{pat}_i; 1 \leq i \leq k \} \quad (3)$$

$$\text{and } \text{cargs} = v_1 \oplus_m \dots \oplus_m v_{n-1} \oplus_m v_n \text{ for } n \geq 1 \quad (4)$$

$$\llbracket \text{match! } \text{expr}_1, \dots, \text{expr}_k \text{ with } \text{ccl}_1 \mid \dots \text{ccl}_l \rrbracket_{\text{cexpr } m} = \text{let } v_1 = \text{expr}_1 \text{ in } \dots \text{let } v_k = \text{expr}_k \text{ in choose}_m [ \llbracket \text{ccl}_1 \rrbracket_{\text{ccl } m, (v_1, \dots, v_k)}; \dots; \llbracket \text{ccl}_l \rrbracket_{\text{ccl } m, (v_1, \dots, v_k)} ] \quad (5)$$

**Figure 3. Translation of monadic match**

matching computation into an ordinary expression that doesn’t contain computation expressions. We extend the translation described in Figure 1 by adding the case for the `match!` construct and we also define rules for translating computation clauses. The Figure 3 shows a rule (1) which translates a computation expression into an ordinary expression. The rest of the rules define the following two translation functions:

$$\begin{aligned} \llbracket - \rrbracket_{\text{cexpr}} &: \text{cexpr} \times \text{ident} \rightarrow \text{expr} \\ \llbracket - \rrbracket_{\text{ccl}} &: \text{ccl} \times \text{ident} \times [\text{ident}] \rightarrow \text{expr} \end{aligned}$$

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.

In the translation rules,  $\text{bind}_m$  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  $m.\text{Bind}$ ).

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  $\text{choose}_m$  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  $\oplus_m$  (4). The operator is left-associative, so when combining for example three values, the resulting value will be of type  $M\langle ('a * 'b) * 'c \rangle$ .

Finally, we pass the combined monadic value as an argument to a  $\text{map}_m$  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  $M\langle \text{MaybeDelayed}\langle 'a \rangle \rangle$ .

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 monadic value. Allowing this case would require a special handling and it is not clear what the semantics of this clause should be. We discuss this problem further in Appendix C.

## 5. 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*.

### 5.1 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:

$$\text{val } \oplus : M\langle 'a \rangle \rightarrow M\langle 'b \rangle \rightarrow M\langle 'a * 'b \rangle$$

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:

$$\begin{aligned} \text{let } \text{assoc } ((a, b), c) &= (a, (b, c)) \\ \text{let } \text{swap } (a, b) &= (a, b) \end{aligned}$$

$$\text{return } (a, b) \equiv (\text{return } a) \oplus (\text{return } b) \quad (C1)$$

$$u \oplus (v \oplus w) \equiv \text{map } \text{assoc } ((u \oplus v) \oplus w) \quad (C2)$$

$$u \oplus v \equiv \text{map } \text{swap } (v \oplus u) \quad (C3)$$

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 [25], which relates parallel composition operator (\*\*\*) in place of our  $\oplus$  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. More information is provided in the Appendix A.

The law resembling commutativity also reveals interesting relations between our *merge* operation and the *bind* operation in commutative monads...

## 5.2 Merging in commutative monads

When introducing *merge* in section 3.1, 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:

```
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 tuple values. 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 1<sup>st</sup> value of *ma* and for example 3<sup>rd</sup> value of *mb*. Waiting in the reversed order can give completely different results.

Now we know that we cannot use this straightforward implementation blindly. However, we'll see that it obeys the *merge* laws in one important special case.

**Commutative monads.** It is a well known fact that every monad should obey the three *monad laws* that are described in [2]. We will need these rules shortly, so we show them here for convenience (we denote the *bind* operation using "*>>=*"):

```
return a >>= f ≡ f a           (Left id.)
m >>= return ≡ m              (Right id.)
(m >>= f) >>= g ≡ m >>= (λx → f x >>= g) (Assoc.)
```

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<sup>1</sup>:

```
m { let! a = <expr>1
    let! b = <expr>2
    <cexpr> } ≡ m { let! b = <expr>2
                  let! a = <expr>1
                  <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 [15], 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:

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

For commutative monads, the above code gives a correct definition of the *merge* operation (a proof is presented in Appendix D). 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 [15]). 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!*):

```
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.

<sup>1</sup> F# doesn't use monads to control side-effects, so the expressions *<expr><sub>1</sub>* and *<expr><sub>2</sub>* may contain side-effects, which would be indeed reversed. We focus only on effects that is controlled by the monad.

### 5.3 Merging and applicative functors

*Applicative functors* (also called *idioms*) [16] 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:

```
map assoc (u * (v * w)) ≡ (u * v) * w      (Associat.)
map (f × g) (u * v)    ≡ map f u * map g v  (Natural.)
map snd (unit * v)     ≡ v                  (Left id.)
map fst (u * unit)     ≡ u                  (Right id.)
```

The  $*$  operation must obey the *associativity law*, which we also required for  $\mathbb{D}$ . The other laws are different and are not equivalent. As discussed in [16], 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  $\mathbb{D}$ . 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 { let! f = fs
                  match! fs, xs with
                    | !f, !x -> return f x }
```

Figure 4. Defining applicative functors

**Defining application.** For any monad, we can implement the  $\otimes$  operation using *bind*. This is always a valid definition and it is shown in Figure 4 (left). If we have a monad with the *merge* operation, we can implement the  $\otimes$  operation in terms of *merge*. Figure 4 (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 [16] to define  $\otimes$  in terms of  $*$  when showing that the two formulations of applicative functors are equivalent.

This way of defining  $\otimes$  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  $\otimes$  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  $\otimes$  in terms of *merge* gives an alternative applicative functor that can be also useful.

We'll look at Imperative streams [17], 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  $\otimes$ , 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  $\otimes$  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 code looks like this:

```
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) ->
5:     match! xs, trans xss with
6:     | !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 5). 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 6).

Even though the structure of the two examples isn't exactly the same, there are similarities. If we expanded the use of  $\otimes$  in the first version to the definition using `match!` shown in Figure 4 (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  $\otimes$  operation). This gives F# computation expressions with our extension an additional and very useful expressive power.

## 6. 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

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

Figure 5. Monad laws formulated using `join` and `choose`

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.

### 6.1 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(\_ -> expr1) m);
         map (\x -> Success(\_ -> expr2) m) ]
≡ map (\x -> Success(\_ -> expr1) 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.

## 6.2 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

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

definition that uses `join`, `map` and `return` from [20]. The laws are shown in Figure 5 (left).

To get a new set of laws, we could simply replace `join` with `choose · L · (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 5 (right).

## 7. 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`:

$$\text{match! } m \text{ with } \quad \equiv \quad \text{let! } var = m$$

```
!var -> expr          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!  $m_{p(1)}, \dots, m_{p(n)}$  with
|  $cpat_{1, p(1)}, \dots, cpat_{1, p(n)}$  ->  $cexpr_1$  | ...
|  $cpat_{k, p(1)}, \dots, cpat_{k, p(n)}$  ->  $cexpr_k$ 

```

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    $\equiv$    match  $e_1, e_2$  with
|  $!var_1, !var_2$  ->  $cexpr$       |  $var_1, var_2$  ->  $cexpr$ 

```

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 (\(var1, var2) →  $cexpr$ ) (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!  $m_1, \dots, m_n$  with      match!  $m_1, \dots, m_n, m$  with
|  $c_{1,1}, \dots, c_{1,n}$  ->  $cexpr_1$     $\equiv$    |  $c_{1,1}, \dots, c_{1,n}, \_$  ->  $cexpr_1$ 
| ...                               | ...
|  $c_{k,1}, \dots, c_{k,n}$  ->  $cexpr_k$    |  $c_{k,1}, \dots, c_{k,n}, \_$  ->  $cexpr_k$ 

```

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  $v$   $\equiv m$ 

```

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 5 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
|  $!var_1$  ->  $\langle cexpr \rangle_1$     $\equiv$    match!  $m$  with
|  $!var_1$  ->  $\langle cexpr \rangle_1$ 
|  $!var_2$  ->  $\langle cexpr \rangle_2$ 

```

This guarantees that the intuition about unreachable clauses is, to some extent, also valid for the *match!* construct. The equivalence is a direct 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.

## 8. 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.

### 8.1 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.

Some monads provide a way to represent failure inside the monad. For example, a failure in the `List` monad is an empty list `[]`. This opens a question whether we could represent a failure inside the monad. 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 works well for us:

- **Inner value.** This means that the body of the clause evaluates to a failure value inside the monad. However, in this case, the body may fail *later*, which means that we run a part of one clause and then continue running another clause.
- **Outer value.** In this case, we'd modify the translation to use `bind` instead of `map` in Figure 3 (2). This, however, changes the structure of the outer monad, which makes it impossible to implement the *choose* operation for some monads.

A curious reader can consult Appendix B, which provides some more details about the problematic cases. However, for a language without side-effects, the representation of failure in the inner value may be very well reasonable.

### 8.2 Monadic Active Patterns

Active patterns [13] were introduced as a mechanism for creating abstractions over algebraic data types. This has been a well-understood problem since [14]. 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 [13]. However, [13] also proposes 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 `MonadPlus` 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.

### 8.3 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 Appendix C. It also clarifies why we cannot handle a case when no value is available (in the second type of monad).

### 8.4 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 (for example 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 construc-

ted 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 need 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 paper we focused on a 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.

## 9 Related work and conclusions

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.

**Computation types.** Apart from monads [2], there are several other types of computation models. Applicative functors [16] 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. Comonads (a categorical dual of monads) [28] have shown useful for data-flow programming [30].

Arrows [22, 25] 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 [24] and we believe it may be interesting to consider whether generalized pattern matching could be provided in the notation as well.

**Pattern matching.** The work on pattern matching mainly focused on providing better abstraction when pattern matching on normal values [14, 21]. 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 [29, 13] as discussed in section 8.2. 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 [26] 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 paper.

**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 [10]. 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 [27]. However, as noted in [7], 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 [23]) and uses imperative programming encoded using monads (similarly to [17]). We also presented parallel programming model based on futures [11]. Pattern matching in this model is very similar to the `pcase` (parallel case) introduced in Manticore [12].

## 9.1 Conclusions

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 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.

## Acknowledgements

We thank to Simon Peyton Jones, Gregory Neverov and James Margetson for useful comments and suggestions about this work. Tomas is also grateful to Microsoft Research for inviting him to an internship, which made this work possible.

## References

- [1] D. Syme, A. Granicz, and A. Cisternino. *Expert F#, Introducing Language-oriented Programming*. Apress, 2007.
- [2] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, LNCS 925, 1995.
- [3] Microsoft. F# Language Specification. Available online at: <http://tinyurl.com/fsspec>, Retrieved February 2010
- [4] D. Syme, A. Granicz, and A. Cisternino. *Expert F#, Reactive, Asynchronous and Concurrent Programming*. Apress, 2007.
- [5] C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the join- calculus. In *Proc. POPL 1996*.
- [6] 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.
- [7] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
- [8] C. Russo. The Joins concurrency library. In *PADL 2007*.
- [9] S. Singh. Higher-order combinators for join patterns using STM. In *Proc. TRANSACT Workshop, OOPSLA, 2006*.
- [10] P. Haller, T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. In *Proc. COORDINATION 2008*.
- [11] H. Baker, C. Hewitt. "The Incremental Garbage Collection of Processes". In *Proc. Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12*.
- [12] M. Fluet, M. Rainey, J. Reppy and A. Shaw. Implicitly-threaded parallelism in Manticore. In *Proc. ICFP 2008*
- [13] Syme, D., G. Neverov, J. Margetson. Extensible Pattern Matching via a Lightweight Language Extension. *ICFP, 2007*.
- [14] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. POPL, 1987*
- [15] S. P. Jones. Wearing the hair shirt - A retrospective on Haskell. Invited talk, *POPL 2003*. Slides available online at: <http://tinyurl.com/haskellretr>
- [16] McBride, C. and R. Paterson, *Applicative programming with effects*, *Journal of Functional Programming* 18 (2008)
- [17] E. Scholz. Imperative streams - a monadic combinator library for synchronous programming. In *Proc. ICFP, 1998*
- [18] S. P. Jones (ed.) *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
- [19] 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*.
- [20] D. King, P. Wadler. Combining Monads. In *Proc. of Glasgow Workshop on Functional Programming, 1992*.
- [21] C. Okasaki. Views for Standard ML. In *Proc. Workshop on ML, Baltimore, Maryland, USA, pp. 14–23, 1998*.

- [22] J. Hughes, Generalising Monads to Arrows, in Science of Computer Programming 37, pp67-111, May 2000.
- [23] C. Elliott, Declarative event-oriented programming. In Proceedings of PPDP 2000
- [24] R. Paterson. A new notation for arrows. In ICFP 2001
- [25] Hai Liu. E. Cheng. P. Hudak. Causal Commutative Arrows and Their Optimization. In Proc. ICFP 2009
- [26] B. Emir, Odersky, M., Williams, J. Matching Objects with Patterns. In ECOOP 2007.
- [27] Ma Qin, L. Maranget. Compiling Pattern-Matching in Join-Patterns, In Proc. CONCUR 2004
- [28] R. Kieburtz. Codata and Comonads in Haskell. Unpublished draft, 1999. <http://tinyurl.com/comonads>
- [29] M. Tullsen. First class patterns. In Proc. PADL, 2000
- [30] T. Uustalu, V. Vene. The essence of dataflow programming. In Proceedings of APLAS 2005

## Appendix A: 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 [18] use Cartesian product semantics, but Data Parallel Haskell [19] uses zip semantics.

As discussed in section 5.1, 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.

## Appendix B: Representing failure inside monad

As discussed in section 8.1, our design represents pattern matching failure explicitly outside of the monadic type using the `Maybe-Delayed<'a>` type. 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. In section 8.1 we also briefly introduced the difficulty with choosing how to represent the failure, in particular whether we should use the inner or outer monad of the value produced by the clause: `M<M<'a>>`.

**Inner value.** If we wanted to represent the failure in the inner computation, we could still use the *map* operation in the translation (Section 4.2). 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.** We argued that pattern matching shouldn't affect the structure of the outer monad in any way. 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 3) 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].

### Appendix C: Compile-time pattern checking

In section 8.3, we draw a distinction between two types of monads. In this appendix, we discuss the compile-time checking that could be provided for these two types.

**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:

$$!<pat> \rightarrow <pat> \quad (\text{binding pat.}) \quad \_ \rightarrow \_ \quad (\text{ignore pat.})$$

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.

**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.

**Ignore all pattern.** The translation semantics from Figure 3 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.

### Appendix D: Merge for commutative monads

In the section 5.2, we discussed an interesting relation between monads that provide the *merge* operation (obeying the laws discussed in section 5.1) 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  $\oplus$  in terms of *bind* and *return*.

This Figure 6 presents a proof that the implementation (shown in section 5.2) 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*):

$$\begin{aligned} & (m \gg= (\lambda a \rightarrow expr)) \gg= g \\ = & \quad (\text{associativity}) \\ & m \gg= (\lambda a \rightarrow ((\lambda a \rightarrow expr) a) \gg= g) \\ = & \quad (\beta \text{ reduction}) \\ & m \gg= (\lambda a \rightarrow expr \gg= g) \end{aligned}$$

**(C1) Merging two returns produces a tuple**

$$\begin{aligned}
& \text{return } a \oplus (\text{return } b) \\
= & \text{ (definition of } \oplus) \\
& (\text{return } a) \gg= (\lambda a \rightarrow \text{return } b \gg= (\lambda b \rightarrow \text{return } (a, b))) \\
= & \text{ (left identity)} \\
& (\text{return } a) \gg= (\lambda a \rightarrow \text{return } (a, b)) \\
= & \text{ (left identity)} \\
& \text{return } (a, b)
\end{aligned}$$
**(C2) Associativity**

$$\begin{aligned}
& \text{map assoc } ((a \oplus b) \oplus c) \\
= & \text{ (definition of map)} \\
& ((a \oplus b) \oplus c) \gg= (\text{return} \cdot \text{assoc}) \\
= & \text{ (definition of } \oplus) \\
& ((a \oplus b) \gg= (\lambda x \rightarrow c \gg= (\lambda c \rightarrow \text{return } (x, c)))) \gg= (\text{return} \cdot \text{assoc}) \\
= & \text{ (definition of } \oplus) \\
& (a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow \text{return } (a, b)))) \gg= (\lambda x \rightarrow c \gg= (\lambda c \rightarrow \text{return } (x, c))) \gg= (\text{return} \cdot \text{assoc}) \\
= & \text{ (associativity, 3x)} \\
& (a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow \text{return } (a, b) \gg= (\lambda x \rightarrow c \gg= (\lambda c \rightarrow \text{return } (x, c))))) \gg= (\text{return} \cdot \text{assoc}) \\
= & \text{ (left identity)} \\
& (a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow c \gg= (\lambda c \rightarrow \text{return } ((a, b), c)))) \gg= (\text{return} \cdot \text{assoc}) \\
= & \text{ (associativity, 4x)} \\
& a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow c \gg= (\lambda c \rightarrow \text{return } ((a, b), c) \gg= (\text{return} \cdot \text{assoc})))) \\
= & \text{ (left identity)} \\
& a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow c \gg= (\lambda c \rightarrow \text{return } (\text{assoc } ((a, b), c)))) \\
= & \text{ (definition of assoc)} \\
& a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow c \gg= (\lambda c \rightarrow \text{return } (a, (b, c))))) \\
= & \text{ (left identity; backwards)} \\
& a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow c \gg= (\lambda c \rightarrow \text{return } (b, c) \gg= (\lambda x \rightarrow \text{return } (a, x))))) \\
= & \text{ (associativity; 2x; backwards)} \\
& a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow c \gg= (\lambda c \rightarrow \text{return } (b, c)) \gg= (\lambda x \rightarrow \text{return } (a, x)))) \\
= & \text{ (definition of } \oplus; \text{ backwards)} \\
& a \oplus (b \gg= (\lambda b \rightarrow c \gg= (\lambda c \rightarrow \text{return } (b, c)))) \\
= & \text{ (definition of } \oplus; \text{ backwards)} \\
& a \oplus (b \oplus c)
\end{aligned}$$
**(C3) Commutativity using laws of commutative monads**

$$\begin{aligned}
& \text{map swap } (b \oplus a) \\
= & \text{ (definition of map)} \\
& (b \oplus a) \gg= (\text{return} \cdot \text{swap}) \\
= & \text{ (definition of } \oplus) \\
& (b \gg= (\lambda b \rightarrow a \gg= (\lambda a \rightarrow \text{return } (b, a)))) \gg= (\text{return} \cdot \text{swap}) \\
= & \text{ (associativity)} \\
& b \gg= (\lambda b \rightarrow (a \gg= (\lambda a \rightarrow \text{return } (b, a))) \gg= (\text{return} \cdot \text{swap})) \\
= & \text{ (associativity)} \\
& b \gg= (\lambda b \rightarrow a \gg= (\lambda a \rightarrow \text{return } (b, a) \gg= (\text{return} \cdot \text{swap}))) \\
= & \text{ (left identity)} \\
& b \gg= (\lambda b \rightarrow a \gg= (\lambda a \rightarrow \text{return } (\text{swap } (b, a)))) \\
= & \text{ (definition of swap)} \\
& b \gg= (\lambda b \rightarrow a \gg= (\lambda a \rightarrow \text{return } (a, b))) \\
= & \text{ (commutativity)} \\
& a \gg= (\lambda a \rightarrow b \gg= (\lambda b \rightarrow \text{return } (a, b))) \\
= & \text{ (definition of } \oplus; \text{ backwards)} \\
& a \oplus b
\end{aligned}$$
**Figure 6. Proof that the default merge for commutative monads is correct**