Pattern matching on monadic computations using Joinads

Tomas Petricek (tp322@cam.ac.uk)

Joinad is an abstract computation type that can be used to encode a wide range of programming models from concurrent, reactive and parallel programming domains. Joinad computations are written using a `match!` construct that can be viewed as ML-style pattern matching on abstract (monadic) values. In a previous work, we defined joinads in terms of two operations, but one of them was complex and difficult to reason about.

In this document, we present a simpler version of joinads. We require that every joinad must also be a monad and we split the original complex operation into the use of monadic bind and simple *choose*. This gives us a definition that is amenable to formal analysis. We define laws that should hold about the operation and discuss similarities between our definition and the `MonadPlus` type class known from Haskell.

We present three sample computations that match the joinad type. The computations allow working with *futures* (computations that eventually produce a value); *time-varying values* (computations whose value changes with time) and *imperative streams* (computations that produce values repeatedly at certain times). We model the three types formally and precisely specify the semantics of joinad operations.

Table of Contents

1. Introducing joinads

Language designers have created many programming models to make it easier to develop programs for numerous specific domains or for particular environments. To name just a few: futures [17] enable running computations in parallel; imperative streams [2] can be used to describe event-based or synchronous computations (e.g. user interfaces or control systems); and behaviors [4] can be used to calculate with values that change over time (e.g. animations).

Unfortunately, writing programs that use these non-standard programming models in a general-purpose language can be difficult. However, extending a language with support for one particular programming model isn't enough, because programmers often need to combine multiple models. We also believe that the choice of a programming model shouldn't affect the choice of programming language.

To address this problem, we designed joinads and the `match!` construct [5]. The general approach is very similar to monads [18]. Just like monads, joinads are abstract computations that can model various concepts (e.g. event, time-varying value or future). Just like the do-notation in Haskell, `match!` construct gives us an easy to use syntax for writing computations that work with joinads. In this document, we refine the notion of joinads to make it more amenable to formal reasoning. The main contributions of this document are:

- We simplify the notion of joinads by requiring that joinad is a monad with three additional simple operations (§1.1) that are easy to reason about.
- We describe several syntactic transformations of `match!` that shouldn't change the meaning of a program (§4.1), we use them to derive laws that should hold about the primitive operations (§4.2) and we prove that the laws are sufficient (§4.3).
- We present three sample computations together with formal semantics that can be used to verify that the computations obey joinad laws (§2). We look at futures (§2.1), time-varying values (§2.2) inspired by behaviors [4] and events (§2.3) inspired by imperative streams [2].
- We discuss how our computation relates to `MonadPlus` type class from Haskell libraries. Our work supports the argument to reform the type class [1], but with a slight modification in the required laws. We describe how our laws differ and how to unify the laws proposed by others and us (§5.1).

In the next section, we introduce joinads as an abstract computation. Then we review our three computation types and discuss reasoning about joinads.

1.1 Joinad computations

Let's start by looking at the revised definition of joinad. A *joinad* is a *monad* that provides three additional operations. The operations must satisfy several laws that will be discussed later (§4). The following listing shows the required operations for both of the types of computations:

Monad operations

unit	: a → m a	(called *unit*)
≫=	: m a → (a → m b) → m b	(called *bind*)

Additional joinad operations

⩔	: m a → m b → m (a, b)	(called *merge*)
⊖	: m a → m a → m a	(called *choose*)
fail	: m a	(called *fail*)

The *merge* operation can be used to compose two computations "in parallel" (where the meaning of "in parallel" depends on the computation type). Judging just from the type signature, we may think that we could implement the operation in terms of *bind* and *return*, but this is not the case. We require some additional laws about the *merge* operation (§4.3). As a result, the obvious definition is correct only for *commutative monads* (§5.2). In our examples, futures and time-varying values are commutative monads, but imperative streams are not.

The *choose* operation allows us to select between two computations. The meaning depends on the computation type. An analogy that we find useful is that the operation creates a computation that returns the "first available" result (again, the precise meaning depends on the computation type). The *fail* operation is used to create a computation that does not contain any value. When choosing between *fail* and another computation *choose* should always select the other computation. This and other requirements are specified using laws (§4.4).

The signatures of *choose* and *fail* are the same as the signatures of *mplus* and *mzero* operations from the `MonadPlus` type class. However, the laws we require are different than the laws of `MonadPlus`. We believe that choosing a different set of laws is important. The behavior of some of the `MonadPlus` implementations doesn't fit well with the `match!` construct we introduce. For example, we expect that only a single clause will be executed for a given value, which isn't always the case with `MonadPlus`). Also, the purpose of our operations is somewhat different than what the name "plus" suggests.

Interestingly, the laws we require about operations *fail* and *choose* are closely related as those described in the discussion about `MonadOr` type class. This type class has been introduced in the `MonadPlus` reform proposal [1] in order to distinguish different uses of `MonadPlus` (which is presently used for multiple different purposes with different laws). We discuss this topic in more details later (§5).

2. Example joinad computations

In this section, we'll briefly explore three different types of computations that provide the five operations described above. We show a practical example using our language extension for joinads – the `match!` construct, we look how the code is translated in terms of the primitive operations and we give semantics of the five joinad operations. We explain the `match!` construct as we use it in the examples. The syntax and translation rules are given later (§3). Our first two examples are commutative monads.

2.1 Computation for Futures

A future of type `Future<T>` represents a computation that eventually produces a value of type `T`. The bind operation runs a continuation when a value becomes available. The computation is commutative – when we wait for two futures in a sequence, the continuation will be called when both complete (regardless of the binding order).

Example: Multiplying Futures

In this example, we write a function `multiply` that takes two futures and returns a future that will eventually produce a multiple of the values produced by the two given futures. When one of the futures produces 0, we can short-circuit and return 0 immediately. This can be encoded as follows (the syntax is very similar to the `pcase` construct in Manticore [3]):

```
let multiply f1 f2 = future {
  match! f1, f2 with
  | !0, _  -> return 0
  | _, !0  -> return 0
  | !a, !b -> return a * b }
```

The patterns in clauses of the `match!` construct are special syntactic category that we call *computation patterns*. A computation pattern is either "!<*pat*>" (binding pattern) where <*pat*> stands for any standard ML pattern or "!_" (ignore pattern). An ignore pattern specifies that we don't need to obtain value of the corresponding computation in order to run the body of the clause. In the example above, this means that when the first future completes returning zero, a clause can be executed without waiting for the other future.

Translation using joinad operations

The following listing shows translation of the previous example in terms of primitive operations. The detailed translation rules are given in §3.

```
let multiply f1 f2 =
  ( (f1 >>= function 0 -> unit(unit 0) | _ -> fail) ⊖
    (f2 >>= function 0 -> unit(unit 0) | _ -> fail) ⊖
    ((f1 ⓫ f2) >>= function (a, b) -> unit(unit a * b)) ) >>= id
```

When desugaring the `match!` construct, the compiler processes each clause independently. We'll get to the aggregation of clauses later and focus on individual clauses first. To translate a clause, we first identify which of the computations passed as arguments to `match!` are matched against binding pattern. We need to obtain a value from all of these computations, which is done by merging them using the *merge* (⓫) operation.

The result of *merge* is passed to monadic *bind*. In the continuation, we match the value (or a tuple generated by *merge*) against the actual patterns used in the clause. The body of the function always returns a computation created using *unit* or *fail*. This is important for some more complex types of computations (§2.3) and it means that we can require weaker laws for the *choose* (⊖) operation (§4.4).

If the pattern matching succeeds, the function wraps the translated body of the clause using the *unit* operation. In our case, the translated body is `unit 0` and `unit a * b` respectively and we wrap it using another *unit*. (In more complicated examples, the body may contain binding using >>= or other calls.) In case of failure, the function returns a computation representing a failure (using the *fail* operation).

A translated clause in our example has a type `Future<Future<int>>`. Using the *choose* (⊖) operation, we aggregate the clauses into a single value of the same type. To turn this value into a simple computation (`Future<int>`), we need to unwrap the result and "run the inner computation inside the outer computation". This is done by adding monadic *bind* with identity function to the end of the expression. (This corresponds to calling the *join* operation from an alternative definition of monads.)

Semantics for Futures

A future can be modeled as a pair containing time (when the future completed) and a value (the produced result). In this example, the time is continuous and the only way to create two futures that complete at the same time is to use *unit* (which creates a future that has a value immediately available). The following listing presents semantics of the five joinad operations for futures.

$$[\![-]\!] \;:\; \text{expr} \to \text{time} \to \text{time} \times \text{value}$$

$$\begin{aligned}
[\![\text{unit } v]\!]_t &= (t, v) \\
[\![\text{fail}]\!]_t &= (\infty, \bot) \\
[\![e_1 \text{ ⓫ } e_2]\!]_t &= \big(\max\{t_1, t_2\}, (v_1, v_2)\big) \text{ where } (t_i, v_i) = [\![e_i]\!]_t \text{ for } i \in \{1, 2\} \\
[\![e_1 \text{ ⊖ } e_2]\!]_t &= \begin{cases} (t_1, v_1), & t_1 \le t_2 \\ (t_2, v_2), & t_1 > t_2 \end{cases} \quad \text{where } (t_i, v_i) = [\![e_i]\!]_t \text{ for } i \in \{1, 2\} \\
[\![e \gg\!= f]\!]_t &= [\![f\ v_1]\!]_{t_1} \text{ where } (t_1, v_1) = [\![e]\!]_t
\end{aligned}$$

The meaning of an expression is defined as a tuple consisting of a value and a time when it is produced. It is parameterized by the time when the execution of the expression starts. The first two operations are simple – *unit* creates a future that produces the given value immediately and *fail* creates a future that never produces a value.

The *merge* (⏸) and *choose* (⊜) operations are in some way orthogonal. *Merge* waits until both of the futures produce a value, so it completes when the later one completes and returns a tuple containing both values. *Choose* waits for the first of the two futures and returns the value that was available as the first one.

The *bind* operation is similarly straightforward. It takes the value produced by the input future, evaluates the continuation (to get a future) and returns the meaning of this created future. The evaluation of a function is not defined in this document, but could be provided in the usual way. Note that our simplified semantics assumes that the evaluation of the function terminates and does not take any time. In order to create some interesting futures, we also need a primitive that introduces a delay:

$$[\![\text{after } d\ v]\!]_t \quad = \quad (t + d + \delta, v)$$

The primitive `after` creates a future that produces a given value after a specified time. To model non-determinism in the model, we also add a small random δ. In the next section, we'll look at our second sample computation, which is inspired by one of the concepts from Functional Reactive Programming.

2.2 Computation for Time-varying values

A time-varying value is a type of computation used in Functional Reactive Programming [4] (where it is called a *behavior*). It represents a value that is changing in time. How a time-varying value constructed using the *bind* operation behaves? When we get its value at a certain time, it gets value of the input time-varying value, evaluates the continuation (as with the previous model, we assume this takes no time) and gets the current value of that computation.

In order to implement the *fail* operation, we need to slightly extend the simple notion of time-varying values from FRP. At any time, the computation may produce a value or a value may be unavailable. This means that instead of using `TimeVarying<T>`, we work with `TimeVarying<option<T>>`. The following section demonstrates how we can write time-varying computations using `match!`

Example: Time-varying condition

This example assumes that we have two time-varying values of type `Changing<int>`. We want to construct a time-varying computation with a value `true` when the value of the first argument is smaller than the second one and a value `false` otherwise. Using `match!` we can write the following:

```
let smaller t1 t2 = changing {
  match! t1, t2 with
  | !u, !v when u < v -> return true
  | !_, !_ -> return false }
```

The first clause matches when both of the changing values are defined and the value of the first one is larger than the value of the second one. In the second clause, we again require value from both of the computations (by using binding pattern "!<*pat*>"), but we ignore the produced values and simply return `false`. Note that the order of clauses is important in this example – any value matched by the first clause would be also matched by the second clause. The sensitivity to the order is required by a law about the *choose* operation that we discuss later (§4.2).

Translation using joinad operations

Both of the clauses contain two binding patterns meaning that we require value from both of the computations. In the translation, this means that both clauses first merge the two time-varying values as follows:

```
let smaller t1 t2 =
  ((t1 ⫫ t2) >>= function
      (u, v) when u < v -> unit(unit true) | _ -> fail) ⫏
  ((t1 ⫫ t2) >>= function (_, _) -> unit(unit false))
```

The first clause doesn't fail when the obtained values match the condition specified by `when`. The `when` guard can be simply copied from the original clause, because the variables in scope are the same as in the original clause. The second clause matches any value and never fails. The automatic translation would also add a redundant case (returning `fail`) that can never match in this example, so we omitted it for simplicity.

Note that the second clause matches only when a value is available in both of the input values (because it contains two binding patterns). When any of the two inputs is undefined at a certain time, the resulting value will be also undefined. Our translation doesn't support clauses that do not contain any binding patterns, although it may be possible to extend the translation to handle this case (§6).

Semantics for Time-varying values

Let's now look at semantics that specifies behavior of the joinad operations. We model time-varying value as a function that takes a time and returns a value at that time. In order to implement the `fail` operation, the function may also return a special value • representing that a computation doesn't have a value. We write $A^{\bullet}$ to stand for $A \cup \{\bullet\}$.

$$[\![-]\!] \;:\; \text{expr} \to (\text{time} \to \text{value}^{\bullet})$$

$$\begin{aligned}
[\![\text{unit } v]\!]_t &= v \\
[\![\text{fail}]\!]_t &= \bullet \\
[\![e_1 \mathbin{⫫} e_2]\!]_t &= ([\![e_1]\!]_t, [\![e_2]\!]_t) \\
[\![e_1 \mathbin{⫏} e_2]\!]_t &= \begin{cases} v_1, & v_1 \neq \bullet \\ v_2, & \text{otherwise} \end{cases} \quad \text{where } v_i = [\![e_i]\!]_t \text{ for } i \in \{1, 2\} \\
[\![e >\!\!>= f]\!]_t &= [\![f\ [\![e]\!]_t]\!]_t
\end{aligned}$$

The functions representing computations created by *unit* and *fail* return the same value regardless of the time. In case of *unit*, the result is always the specified value and in case of *fail*, the result is always • representing an undefined value.

The *merge* operation needs to obtain values from both of the computations. When given a time *t*, it simply gets value of the two merged time-varying values at that time (as for any commutative monad, this can be implemented using *bind*). The *choose* operation returns a value of the first computation (at the current time) if the value is defined. If the first value is not defined, then it returns the current value of the second computation.

Finally, to get the value of a computation created by the *bind* operation, we need to get the value of the time-varying input, then call the continuation function and finally get the value of the returned computation. The definition of the function evaluation is again standard. Just like in the previous example, we need some other primitive for creating computations that actually take the time into account. For example:

$$[\![\text{time}]\!]_t = \text{float}(t)$$

In the following section we'll look at an example that uses the `match!` syntax for working with time-varying computations and discuss the translation.

2.3 Computation for Events

Imperative streams developed by Scholz are a "generalization of the IO monad suitable for synchronous concurrent programming" [3]. An imperative stream `ImpStream<T>` is a computation that produces values of type `T` at certain times. The time is discrete meaning that two streams can produce value at the same time. Imperative streams can have side effects that also occur at specific (discrete) times. In our example, we'll work with simplified model `Event<T>` which doesn't have side-effects and only produces values at (discrete) times.

Informal introduction

Before we look at our first example, we'll briefly and informally explain how two of the joinad operations work. We'll get to the details later when we present the semantics. There are several design options for implementing *bind* and *merge*. The design we select is the following:

- An event created by *merge* produces value only when two events produce a value at exactly the same time. This happens in synchronous systems (with a single global timer), but doesn't happen when events are serialized (e.g. user interface systems based on event loop).
- If we use *unit* as a continuation in the *bind* operation, the created event will occur at the same time as the source event of binding. This means that we can use *bind* to create event that happens at the same time as another event.

To demonstrate the two aspects, let's look at the following example:

```
let e1 = unit 42
let e2 = e1 >>= fun _ -> unit "Hello world"
let merged = e1 ⫼ e2
```

When we start the program (at a certain time), the events `e1` and `e2` will occur and will produce values 42 and "Hello world" respectively. The events will happen in the same "time slot" as when we started the program. As a result, the merged event `merged` will also occur producing a value (42, "Hello world").

Using the *bind* operation to create event that occurs at the same time as another pattern is common programming pattern when working with events. The upcoming section demonstrates this using a larger example.

Example: Reporting signal state

Let's say that we want to implement application where we can create new objects by clicking and remove objects by clicking with the Shift key pressed. When the `click` event occurs, we do not have any way of finding out whether Shift is pressed or not.

To solve this problem, we want to create a new event that will occur at the same time as `click` and will contain Boolean value specifying whether Shift is pressed. We can use events `shiftDown` and `shiftUp` that are triggered when the state of Shift key changes and an event `load` that is triggered when the application starts. The following function implements the pattern:

```
let withState (init, on, off) target = event {
  match! init, on, off with
  | !_, _, _ -> let! _ = target in return false
  | _, !_, _ -> let! _ = target in return true
  | _, _, !_ -> let! _ = target in return false }
```

The function takes a tuple of three `Event<T>` values that specify initial load event and two events that switch the state. The `target` event defines when the created event should produce values. The behavior is implemented using three similar clauses. When

a clause starts running, it uses `let!` and `return` to generate Boolean value whenever the `target` event occurs.

When the initial event occurs, the match! construct selects the first clause and starts emitting `false` values. Later on, when the `on` event occurs, the second clause is selected. As a result, the resulting event starts emitting values generated by the second clause (discarding the event value generated earlier by the first clause). When the `off` event occurs, the emitted value is switched back to `false`. The following snippet shows desugaring of the function:

```
let withState (init, on, off) target =
  ((on   >>= fun _ ->
        unit(source >>= fun _ -> unit(true)))  ⊖
   (off  >>= fun _ ->
        unit(source >>= fun _ -> unit(false))) ⊖
   (init >>= fun _ ->
        unit(source >>= fun _ -> unit(false)))) >>= id
```

Each of the clauses binds only on one of the events, so the translation doesn't need to combine input events using *merge*. In the continuation, we return a translated body of the clause wrapped using the *unit* operation. As in the previous examples, we omitted the *fail* case, because the patterns in this example are exhaustive.

The core part of the behavior of this example is implemented by the *choose* operator. When any of the three events occurs, the continuation of *bind* is executed and it generates an event that happens once carrying the body to be executed. The event returned by *choose* will carry these bodies. When we later flatten the type using "≫= id" we'll create event that emits values generated by bodies of clauses, but changes the value when another clause is selected.

To present a complete demo of working with the *event* computation, we'll look at a computation that uses `withState` to handle clicks with and without the Shift key.

Example: Handling clicks with and without Shift

In this section we'll finish an example that we introduced earlier. We're creating an application where the user can create objects by clicking and delete objects by clicking with the Shift key pressed. We'll implement the core of the processing – an event that yields `Create(x, y)` and `Delete(x, y)` commands.

To implement the desired behavior, we use `withState` to create event carrying reporting the status of the Shift key whenever `click` occurs. We can then bind on the two events using `match!` (because they happen at the same time) and use patterns to generate different commands depending on the Shift status:

```
let on   = filter ((=) Keys.ShiftKey) keyDown
let off  = filter ((=) Keys.ShiftKey) keyUp

let commands = event {
  match! withState (load, on, off) click, click with
  | !false, !(x, y) -> return Create(x, y)
  | !true,  !(x, y) -> return Delete(x, y)
```

We first create two events that are triggered when the Shift key is pressed and released (the `filter` function can be easily implemented using `let!`, `if` and `return`). Next, we use `match!` with two clauses to generate two different commands. The clauses use non-exhaustive patterns. The first one will be called only when the user clicks while the Shift key is not pressed. The desugaring of the example looks as follows:

```
let commands =
  let arg1 = withState (load, on, off) click
```

```
( ((arg1 ⓘ click) ≫=
      function (true, (x, y)) -> unit(unit(Delete(x, y)))
            | _ -> fail) ⊖
  ((arg1 ⓘ click) ≫=
      function (false, (x, y)) -> unit(unit(Create(x, y)))
            | _ -> fail)) ≫= id
```

The first argument to the `match!` construct was an expression (a function call). When translating the construct, we assign the result to a fresh variable. In F#, this prevents executing side-effects repeatedly. The rest of the desugaring follows the same pattern as our earlier examples. Since the patterns in individual clauses are not exhaustive, we included the case that uses *fail* operation when a pattern is not matched. As the laws about *choose* specify (§4.4), it should always select non-failing computation.

Semantics for Event computations

The meaning of joinad operations for events is parameterized by the time when the event is created. The result is an (ordered) set of time-value pairs specifying what values were produced at which time. We use sets instead of lists or sequences because we allow event to happen only once at a given time and because it makes the specification a bit more readable.

$$[\![-]\!] \;:\; \text{expr} \to \text{time} \to [\text{time} \times \text{value}]$$

$$\begin{aligned}
[\![\text{unit } v]\!]_t &= \{(t, v)\} \\
[\![\text{fail}]\!]_t &= \emptyset \\
[\![\text{merge } e_1\ e_2]\!]_t &= \{t_1, (v_1, v_2) | (t_1, v_1) \in [\![e_1]\!]_t, (t_2, v_2) \in [\![e_2]\!]_t, t_1 = t_2\} \\
[\![\text{bind } f\ e]\!]_t &= \left\{(t', v') \in [\![f\ v_1]\!]_{t_1} \restriction_{e,t_1} \middle| (t_1, v_1) \in [\![e]\!]_t\right\} \\
[\![\text{choose } e_1\ e_2]\!]_t &= [\![e_1]\!]_t \cup \{(t_1, v_1) | (t_1, v_1) \in [\![e_2]\!]_t, \nexists v' \colon (t_1, v') \in [\![e_1]\!]_t\}
\end{aligned}$$

$$\text{where } E \restriction_{e,t} = \{(t_1, v_1) \in E | t_1 < \min\{t' | (t', v') \in [\![e]\!]_t, t' > t\}\}$$

The *unit* operation creates an event that occurs once at the time when it was created producing a given value. The behavior of *fail* is more similar to what we've seen earlier for futures. It creates an event that never produces any value, which is represented as an empty set.

The *merge* operation is more interesting. It creates event that occurs only when the two events given as arguments occur at exactly the same time (in that case, the resulting event will contain a tuple with a pair of the values produced by the two events). As already mentioned, events can occur at the same time for example in synchronous systems. In other systems, we can use *merge* to work with events that we constructed.

The *choose* operation combines two events. It takes all occurrences of the first one and adds those occurrences from the second one that do not happen at the same time as some occurrences of the first one. To state it in another way, at every time we return the value produced by the first event (if any) or the second event (if any) preferring the first one if both are available. If none of the events produces the value then we also do not produce any value.

Finally, the *bind* operation is interesting. We define the operation in the same way as Scholz [3]. For each occurrence of the source event *e*, we get an event constructed by the function *f* and return time value pairs defined by this event. However, we don't return all pairs – we restrict the result only to those occurrences that happen before the original event *e* produces another value. After that happens, the first generated event is abandoned and we continue returning values from the second generated event. Formally, this is written using restriction $E \restriction_{e,t}$ which creates an event that produces all occurrences from *E* that happen before the first event of *e* after the time *t*.

3. Language extension

In this section, we give the syntax of the `match!` construct and we formally define the translation to joinad operations. The syntax is the same as in our previous work [5]. The structure of the translation is similar, but it uses the simplified version of joinads.

3.1 Syntax of match!

The extension adds a single new case to the syntactic category of computation expressions defined in [3]. The `match!` construct takes one or more expressions as arguments and has one or more computation clauses.

cpat	=	_	Ignore pattern
		!pat	Binding pattern
ccl	=	$cpat_1$, …, $cpat_k \rightarrow cexpr$	Computation match clause
cexpr	=	**match!** $expr_1$, …, $expr_k$ **with**	Computation pattern matching
		ccl_1 | … | ccl_p	…consisting of several clauses

Clauses are formed by computation patterns (*cpat*) with computation expression (*cexpr*) as the body, so we define them as a separate syntactic category for clauses (*ccl*). A computation pattern can be either an ignore pattern (written as "_") or a binding pattern, which is a standard F# pattern [3] prefixed with "!".

3.2 Translation

We extend the translation from the F# specification [6] by adding a case for `match!` The translation is defined in terms of three functions. The first one translates an expression into an expression that does not contain computation expressions. The next two deal with the body of a computation expression and with a computation clause respectively:

$[\![-]\!]$: expr → expr
$\langle\!\langle - \rangle\!\rangle$: cexpr → ident → expr
$\langle - \rangle$: ccl → ident × [ident] → expr

Computation expressions are wrapped in blocks denoted by an expression. The result of this expression is a *computation builder*, which exposes operations defining the computation. In the translation, we pass the builder to functions as an identifier and we write op_m to denote joinad operation *op* provided by the builder *m*. Arguments of `match!` are stored in fresh variables that are passed around as a list of identifiers.

$$[\![expr\{ cexpr\}]\!] \quad \equiv \quad \textbf{let } m = expr \textbf{ in } \langle\!\langle cexpr \rangle\!\rangle_m$$

$$\langle\!\langle \textbf{match! } expr_1, \ldots, expr_k \textbf{ with } ccl_1 \textbf{ | } \ldots ccl_p \rangle\!\rangle_m \quad \equiv \quad (1)$$
$$\textbf{let } v_1 = expr_1 \textbf{ in } \ldots \textbf{ let } v_k = expr_k \textbf{ in}$$
$$(\langle ccl_1 \rangle_{m,\,(v1,\ldots,vk)} \ominus_m \ldots \ominus_m \langle ccl_{p-1} \rangle_{m,\,(v1,\ldots,vk)} \ominus_m \langle ccl_p \rangle_{m,\,(v1,\ldots,vk)}) \gg\!=_m \mathrm{id} \quad (2)$$

$$\langle\, cpat_1, \ldots, cpat_k \rightarrow cexpr \rangle_{m,\,(v1,\ldots,vk)} \quad \equiv \quad (3)$$
$$cargs \gg\!=_m (\textbf{function } (pat_1, \ldots), pat_n \rightarrow \mathrm{unit}_m \langle\!\langle cexpr \rangle\!\rangle_m \mid \_ \rightarrow \mathrm{fail}_m)$$

$$\textbf{where } \{ (pat_1, v_1), \ldots, (pat_n, v_n) \} = \{ (pat, v_i) \mid cpat_i = !pat; 1 \le i \le k \}$$
$$cargs = v_1 \mathbin{⓫_m} \ldots \mathbin{⓫_m} v_{n-1} \mathbin{⓫_m} v_n \quad \text{for} \quad n \ge 1 \quad (4)$$

To translate `match!` (1), we construct a fresh value for each of the arguments. Then we translate clauses and combine them using *choose* (⊖). The type of computations pro-

duced by individual clauses is `M<M<'T>>`. By applying *choose*, we get the same type, so we use *bind* (≫=) with identity function to flatten the computation type into just `M<'T>`.

To translate a clause (2), we identify which of the arguments are matched against a binding pattern (3). Then we combine all needed computations into a single value (4) using the *merge* operator (⩒). The merged computation is passed as an argument to *bind* (≫=) with a continuation that pattern matches on the actual values. If the matching succeeds, we return the translated body of the clause using *unit.* If the matching fails, we return the *fail* computation.

4. Reasoning about joinads

In this section, we identify laws that should hold about joinad operations. We do that by looking at semantics preserving source code transformations that we want to be able to perform with the `match!` construct. Then we infer laws that need to hold to make these transformations always valid. This section adds several new transformations that were not included the earlier version.

4.1 Semantics preserving transformations

Our `match!` syntax intentionally resembles ML-style pattern matching. This gives users intuition about how they can use the construct. In order to preserve the intuition, we want to be able to perform the refactorings shown in Figure 1. Note that if you drop all extensions related to joinad (e.g. replace `matach!` with plain `match` and `!<pat>` with just `<pat>`), you get valid equations. The

A. **Binding correspondence.** In a degenerated case with just a single clause that always matches, the meaning of `match!` should be the same as the meaning of `let!` This equation is a property of the translation and doesn't imply any additional laws, but it demonstrates an important property.

B. **Reordering.** The equation specifies that we can arbitrarily reorder the arguments and patterns of the `match!` construct. By analyzing the translation, we can see that this only changes the order in which the merge operations are applied to computations, so this equation imposes laws about the *merge* operation.

C. **Match first.** In the ML-style pattern matching, we can write overlapping patterns. A clause can be unreachable if its pattern matches only values that are matched by earlier patterns. This equation specifies similar thing about `match!` It matches only on a single computation, so it talks just about the *choose* operation.

D. **Match non-failing.** The rule specifies that if we include a clause that always fails[1], it will never be selected and it will not affect the pattern matching in any way. This should be the case when the failing pattern follows a matching clause (first equation) as well as when it precedes it (second equation).

E. **Ignore failed.** If we include a computation constructed using the fail operation as one of the arguments, a clause that matches on it shouldn't have any effect on the result and can be removed. Note that this is the case even when the clause matches on other computations as well. (This is similar to including an empty list and matching only on cons cell in a standard pattern matching.)

F. **Unit match.** This law specifies the behavior of `match!` in a special case when all the arguments are *unit* computations. In that case, we want it to behave as ordinary pattern matching on the values wrapped using *unit*.

[1] A pattern like this can be constructed differently for various types (e.g. `1 & 2` for integers). In F#, we can also write pattern that always fails using active patterns [28].

match! m **with** $!v \to cexpr$	≡	**let!** $v = m$ **in** $cexpr$	(A)
match! $m_{p(1)}, \ldots, m_{p(n)}$ **with** $\mid cpat_{1,p(1)}, \ldots, cpat_{1,p(n)} \to cexpr_1 \mid \ldots$ $\mid cpat_{k,p(1)}, \ldots, cpat_{k,p(n)} \to cexpr_k$		Are equivalent for any permutation p of n numbers	(B)
match! m **with** $\mid !v_1$ -> $cexpr_1$ $\mid !v_2$ -> $cexpr_2$	≡	**match!** m **with** $\mid !v_1$ -> $cexpr_1$	(C)
match! m **with** $\mid !v$ -> $cexpr_1$ $\mid$!Fail -> $cexpr_2$	≡	**match!** m **with** $\mid !v$ -> $cexpr_1$	(D)
match! m **with** $\mid$!Fail -> $cexpr_1$ $\mid !v$-> $cexpr_2$	≡	**match!** m **with** $\mid !v$ -> $cexpr_2$	(D)
match! m, fail **with** $\mid !v_1, \_$ -> $cexpr_1$ $\mid !v_2, !v_3$ -> $cexpr_2$	≡	**match!** m **with** $\mid !v_1$ -> $cexpr_1$	(E)
match! $\text{unit}_m\ e_1, \text{unit}_m\ e_2$ **with** $\mid !v_1, !v_2 \to cexpr$	≡	**match** e_1, e_2 **with** $\mid v_1, v_2 \to cexpr$	(F)
match! $\text{map}_m\ f\ e_1, \text{map}_m\ g\ e_2$ **with** $\mid !v_1, !v_2 \to cexpr$	≡	**match!** e_1, e_2 **with** $\mid !v_1, !v_2 \to cexpr[f\ v_1/v_1, g\ v_2/v_2]$	(G)

Figure 1. Semantics preserving transformations involving `match!`

G. **Map match.** This law is similar to the previous one in the sense that it defines the behavior of `match!` when arguments are created in a special way. It specifies that we can replace pattern matching on computations created using *map* by a pattern matching on the ordinary computations. To do that, we replace occurrences of the variables in the body with corresponding function calls.

In the following sections, we look at the individual joinad operations and the laws that they are required to obey in order to get the equalities above.

4.2 Laws for joinad operations

We briefly review the usual laws that we inherit from the definition of monad. We repeat them mainly to have a reference point when we'll use them later:

$$\text{unit}\ a \gg\!= f \equiv f\ a \qquad \text{(left identity)}$$
$$m \gg\!= \text{unit} \equiv m \qquad \text{(right identity)}$$
$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= \lambda x \to f\ x \gg\!= g \qquad \text{(associativity)}$$

We'll also sometimes use operations from an alternative (but equivalent) definition of monads in terms of *map*, *join* and *unit*. When we use the *map* operation, we use it just as a shorthand for $u \gg\!= \text{unit} \circ f$. Similarly, the join operation is a shorthand for $u \gg\!= \text{id}$.

Now, let's look at our additional laws about joinad operations that are sufficient to make the transformations presented in this section semantics preserving.

Merge operation

The *merge* operation needs to obey commutativity and associativity laws that are common to many computation types. However, we do not choose these laws just to conform to the common practice. Indeed, they are needed to prove one of the earlier equations. The next two laws are less common and define how *merge* behaves with respect to computations created using *unit* or *map*:

$$
\begin{aligned}
u \oplus v &\equiv \text{map swap } (v \oplus u) && \text{(commutativity)}\\
u \oplus (v \oplus w) &\equiv \text{map assoc } ((u \oplus v) \oplus w) && \text{(associativity)}\\
\text{map } (f \times g)\ (u \oplus v) &\equiv (\text{map } f\, u) \oplus (\text{map } g\, v) && \text{(naturality)}\\
\text{unit } (a, b) &\equiv (\text{unit } a) \oplus (\text{unit } b) && \text{(unit merge)}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{where}\quad & \text{assoc } ((a, b), c) = (a, (b, c))\\
& \text{swap } (a, b) = (b, a)\\
& f \times g\, (a, b) = (f\, a, g\, b)
\end{aligned}
$$

The first two laws can be used to arbitrarily rearrange elements of a sequence of computations that is aggregated using the *merge* operation. Together with properties of the translation, this guarantees that the *Reordering* equation (B) will hold. The commutativity law reveals a connection with commutative monads as discussed in our earlier work [5] and later in §5.2.

The *naturality* law specifies how merge behaves with respect to *map*. It can be used to prove the *map match* (G) equation (together with properties of the translation and usual monad laws). *Naturality* and *associativity* are also present in the definition of idioms [16] (we discuss the similarity in more details in §6.2). Similarly, *unit merge* law specifies how the *merge* operation behaves with respect to *unit*. The law is sufficient to prove the *Unit match* (F) equation. It may be of interest that this law has the same structure as the product law of causal commutative arrows [7].

Fail operation

The next operation that a joinad must provide is *fail*. The operation is used to represent a failure using the computation. In our previous work, we represented failure explicitly using the `option` type, but that approach makes reasoning about computations more difficult. The following laws specify how *fail* interacts with other joinad operations:

$$
\begin{aligned}
\text{fail} \gg\!= f &\equiv \text{fail} && \text{(left bind zero)}\\
m \gg\!= \lambda v \to \text{fail} &\equiv \text{fail} && \text{(right bind zero)}\\
u \oplus \text{fail} \equiv \text{fail} \oplus u &\equiv \text{fail} && \text{(merge zero)}
\end{aligned}
$$

The first two laws define the relationship between the *fail* operation and monadic *bind*. The *left zero* law seems to be widely accepted for the `MonadPlus` type class as well [9]. The *right zero* law is slightly controversial (as it may cause troubles when the *m* value is bottom), it is sometimes included in the documentation [10]. We choose it because we're working in an eager language.

The *merge zero* law is a variant of the first two laws for the *merge* operation. As discussed in details later (§5.2), we can implement *merge* for commutative monads in terms of *bind*. Then the merge *zero law* becomes just a consequence of the first two.

Choose operation

The simplified choose operation is the key new idea described in this document when compared to our previous work [5]. The following two laws specify that the operations *fail* and *choose* (⊖) should form a monoid. This is again generally accepted law about the `MonadPlus` type class and we require them as well:

$$
\begin{aligned}
(u \ominus v) \ominus w &\equiv u \ominus (v \ominus w) && \text{(monoid associativity)}\\
u \ominus \text{fail} \equiv \text{fail} \ominus u &\equiv u && \text{(monoid zero)}
\end{aligned}
$$

In addition to monoid laws, we add a law that is novel in our work. We'll discuss how this law relates to a similar law about the `MonadOr` type class later (§5.1).

$$
u \ominus (\text{map } f\, u) \equiv u \qquad \text{(left select)}
$$

The law specifies that if we choose between an original computation (as the first argument) and a computation created using *map* operation, we get a computation equivalent to the one on the left-hand side.

Intuitively, this means that the operation is not symmetric and that it prefers the computation on the left-hand side. The *map* operation preserves the structure of the computation, so the law specifies that when choosing between two computations that have the same structure, the computation on the right-hand side is never used.

It is worth considering whether the set of laws is consistent. The only combination of laws that may appear problematic is *left select* and *monoid zero* (fail $\ominus$ $u \equiv u$). If we substitute *fail* for *u* in the former one we get:

$$\text{fail} \ominus (\text{map } f \text{ fail}) \equiv \text{fail}$$

However, according to *monoid zero*, when the left argument is fail, the computation should be equivalent to the computation on the right-hand side. This is the case only when (*map f fail*) is equivalent to *fail*. This is always the case because of the definition of *map* and the *right zero* law. Let's now look how the equations mentioned earlier follow from the laws discussed in the last two sections.

4.3 Proving equations using laws

In this section, we'll show how to prove the equations involving *fail* and *choose* from section 4.1 using the laws presented in section 4.2. The fact that we'll need all of the laws discussed above justifies our choice of laws.

Proof for the *Match first* equation (C)

If we rewrite the equation specified by *Match first* using the translation rules presented above (§3.2), we get the following:

$$\begin{aligned} & ((m \gg\!= \lambda v \to \text{unit } e_1) \ominus (m \gg\!= \lambda v \to \text{unit } e_2)) \gg\!= \text{id} \\ \equiv\ & (m \gg\!= \lambda v \to \text{unit } e_1) \gg\!= \text{id} \end{aligned}$$

The symbols *m*, e_1, e_2 and *v* in the equation can stand for any expressions (describing a computation) and a variable respectively. To show that the equation holds, let's define:

$$\begin{aligned} \text{tup } a\ b &= (a, b) & f(v, \_) &= v, e_2 \\ \text{snd } (a, b) &= b & m_1 &= \text{map } (\lambda v \to \text{unit } (\text{map } (\text{tup } v)\ e_1)\ m \end{aligned}$$

The two functions *tup* and *snd* are simple helpers for creating a tuple and extracting second member of a tuple. The function *f* takes a tuple whose first element is assigned to a variable *v*. It returns tuple consisting of the variable and the result of e_2 (note that the variable *v* is in scope, so we can evaluate e_2). The m_1 value is obtained by taking *m* as an input and using two nested *map* calls to turn computations of type `M<M<R>>` into computations `M<M<T * R>>` where `T` is the type of *v* and `R` is the type of e_1. Then we can reason as follows:

$$\begin{aligned} & ((m \gg\!= \lambda v \to \text{unit } e_1) \ominus (m \gg\!= \lambda v \to \text{unit } e_2)) \gg\!= \text{id} \\ \equiv\ & m_1 \ominus (m_1 \gg\!= \text{unit} \circ f)) \gg\!= (\text{map snd}) && \text{(transform values)} \\ \equiv\ & m_1 \ominus (\text{map } f\, m_1)) \gg\!= (\text{map snd}) && \text{(by the definition of } \textit{map}\text{)} \\ \equiv\ & m_1 \gg\!= (\text{map snd}) && \text{(by the } \textit{left select} \text{ law)} \\ \equiv\ & (m \gg\!= \lambda v \to \text{unit } e_1) \gg\!= \text{id} && \text{(transform values back)} \end{aligned}$$

The first and the last steps may not be completely obvious. However, we're just applying *map* operations to change the values inside monadic computations without affecting the computation structure. Originally, the computation *m* carries some value and each of the clauses contains calculation that produces some result. After the change, the

computation m_1 carries both the original value and the result of the left expression e_1. The second clause takes first element of the tuple and evaluates e_2. We apply *map* to the overall result to get the result stored in the second element of the constructed tuple. A more formal proof showing that the transformation is correct can be obtained using "free theorems" about the *map* operation using technique described in [11].

The purpose of the transformation is just to transform the equation into a format that matches the structure of the *left select* law. Once we do that, we apply the law to eliminate the second clause.

Proof for the *Match non-failing* equation (D)

The *Match non-failing* equation specifies that clauses that contain patterns that always fail can be removed. We wrote two different versions of the equation – with the failing clause as the first one and as the second one. The following reasoning shows that the first version of the equation holds:

$$
\begin{aligned}
& ((m \gg\!= \lambda v \rightarrow \text{fail}) \ominus (m \gg\!= \lambda v \rightarrow \text{unit } e)) \gg\!= \text{id} \\
\equiv\ & (\text{fail} \ominus (m \gg\!= \lambda v \rightarrow \text{unit } e)) \gg\!= \text{id} && \text{(by } \textit{right zero}\text{)} \\
\equiv\ & (m \gg\!= \lambda v \rightarrow \text{unit } e) \gg\!= \text{id} && \text{(by } \textit{monoid zero}\text{)}
\end{aligned}
$$

The reasoning is straightforward. We first apply the *right zero* law to replace the encoding of a clause that contains failing pattern with the *fail* computation. Then we can use property of monoid to show that the failing computation can be ignored. The other variant of the equation is similar (because *monoid zero* is symmetric).

For simplicity, the equations that we described in the motivation consist both of just two clauses. However, a similar reasoning would work for larger number of clauses as well. Thanks to the symmetry of *monoid zero*, we don't even need the *monoid associativity*. The only reason why we require monoid associativity is that it allows hiding of translation details from the user. It doesn't matter in which order the compiler aggregates the clauses using $\ominus$.

Proof of the *Ignore failed* equation (E)

The *Ignore failed* equation specifies that clauses that pattern match on a set of computations that includes a computation created by the *fail* operation can be removed (together with clauses that match on it). In the translation, this means that fail is merged using $\oplus$ with other computations and then combined with other clauses using $\ominus$. The following reasoning shows that the equation holds:

$$
\begin{aligned}
& ((m \gg\!= \lambda v \rightarrow \text{unit } e_1) \ominus ((m \oplus \text{fail}) \gg\!= \lambda v \rightarrow \text{unit } e_2)) \gg\!= \text{id} \\
\equiv\ & ((m \gg\!= \lambda v \rightarrow \text{unit } e_1) \ominus (\text{fail} \gg\!= \lambda v \rightarrow \text{unit } e_2)) \gg\!= \text{id} && \text{(by } \textit{merge zero}\text{)} \\
\equiv\ & ((m \gg\!= \lambda v \rightarrow \text{unit } e_1) \ominus \text{fail}) \gg\!= \text{id} && \text{(by } \textit{left zero}\text{)} \\
\equiv\ & (m \gg\!= \lambda v \rightarrow \text{unit } e_1) \gg\!= \text{id} && \text{(by } \textit{monoid zero}\text{)}
\end{aligned}
$$

After the translation, we can see that the input computation of the second clause is *fail*. After we apply the *left zero* law, we get *fail* as the second argument of $\ominus$ and then we can remove the clause using *monoid zero*. The equation shows just a simplified case with only a single *fail* argument, but the reasoning could be used for arbitrary number of *fail* computations as arguments.

5. Relations with other computations

In this section, we explore relationships between joinads and other computation types. joinads are monads with several additional operations meaning that every joinad is also a monad. However, there are interesting relations with other computation types…

5.1 MonadOr type class

The `MonadPlus` type class in Haskell has been historically used for different purposes with different sets of laws. The `MonadPlus` reform proposal [1] suggests splitting of the type class into separate classes with clearer purposes and more precisely defined sets of required laws. One of the new type classes discussed in the proposal is `MonadOr`. It consists of the following operations (in addition to usual monad operations):

$\ominus$: m a → m a → m a (called *or-else*)
zero : m a (called *zero*)

The types of the *or-else* and *zero* operations are the same as the types of our *choose* and *fail*. In addition, `MonadOr` requires the following laws[2]:

$$\text{fail} \gg\!= f \equiv \text{fail} \qquad \text{(left bind zero)}$$
$$m \gg\!= \lambda v \rightarrow \text{fail} \equiv \text{fail} \qquad \text{(right bind zero)}$$
$$u \ominus \text{fail} \equiv \text{fail} \ominus u \equiv u \qquad \text{(monoid zero)}$$
$$(u \ominus v) \ominus w \equiv u \ominus (v \ominus w) \qquad \text{(monoid associativity)}$$

In addition to the standard four laws above, there is one additional law that makes the computation interesting and that differentiates `MonadOr` from a reformed `MonadPlus`:

$$(\text{unit } a) \ominus v \equiv (\text{unit } a) \qquad \text{(left catch)}$$

Just like our *left select*, the *left catch* law is asymmetric. Intuitively, it also prefers the computation on the left-hand side, but in a slightly different way. Our law talks about any computation on the left and a computation obtained in a special way (using *map*) on the right, while *left catch* talks about any computation on the right and a computation obtained in a special way (using *unit*) on the left. It isn't difficult to see that none of the laws implies the other:

- **Left select doesn't imply left catch.** The left select law only talks about combining computations with computations obtained in a special way (using *map*). To give a counter example, our *choose* for events obeys left select, but doesn't obey left catch.
- **Left catch doesn't imply left select.** The left catch law only talks about combining *unit* computations with some other. To give a counterexample, we can modify our *choose* operation for continuous values: $[\![e_1 \ominus e_2]\!]_t = [\![e_1]\!]_0$ This version returns computation that has always the same value as was the value of the left argument at the beginning. The modified version obeys left catch but doesn't obey left select.

Informally, we need *left select* instead of *left catch* because it talks about any computations with the same structure (e.g. times when events occur). This is the case because we create computations like this by returning *unit* (when pattern matching succeeds) from the continuation of *bind* operation in the translation.

We've seen that the *left catch* law doesn't quite fit our needs, but our law may not quite fit the needs of some `MonadOr` computations. What could we do if we wanted to include a computation like ours and like `MonadOr` in standard libraries? There are two fair options – we can either require a lower bound or an upper bound of the two laws…

[2] The proposal doesn't mention the *right bind zero* law. However, the law isn't discussed anywhere in the proposal. We include it because it is usually required together with *left bind zero* (if we decide to accept the difficulties with bottom value).

Lower bound
The first law that could replace our *left select* and the *left catch* law is a law that is weaker than both of the laws and is implied by both of them. This means that any computation that obeys *left select* or *left catch* would also obey the following law:

$$(\text{unit } a) \ominus (\text{unit } b) \equiv (\text{unit } a) \qquad \text{(weak left catch)}$$

Showing that each of the other laws implies *weak left catch* isn't difficult (we just substitute a computation on one side of $\ominus$ with *unit*). This law isn't enough to prove that the *match first* (C) equation holds. We can only show weaker form where the argument for `match!` is a computation created using *unit*. We believe that this weaker form doesn't fully convey the intuition about `match!` especially for computations that produce multiple values (e.g. events), but that could be specified as additional requirement, specific to the `match!` construct.

Upper bound
Another option is to replace *left select* and *left catch* with a law that is stronger than both of the laws. The new law holds only when both of the original laws hold. To write the law, it is sufficient to replace the use of *map* in *left select* with binding as follows:

$$u \ominus (u \gg\!= \mathrm{g}) \equiv u \qquad \text{(strong left select)}$$

We already said that our computation for events doesn't obey the left *catch law*, so it also cannot obey the above law. Whether we can provide an alternative behavior of the *choose* operation for events that would obey this law (as well as all other required laws) is a question that remains to be answered.

The *strong left select* law holds for other computations used in this document (futures and time-varying values) as well as for `MonadOr` computations described in [1] (maybe monad, IO and `MonadOr` instance for lists, where *choose* returns the first non-empty argument).

When introducing *left select*, we checked whether it is consistent with other laws and in particular with *monoid zero*. The stronger version of the law is still consistent using exactly the same reasoning. For a computation that obeys the *strong left select* law, we can derive the two weaker laws as follows:

- To prove that the *left catch* law holds, we define $f = \lambda x \to v$. We start with the left-hand side of the law and show that it equals the right-hand side as follows:

$$\begin{aligned}
& (\text{unit } a) \ominus v \\
\equiv\ & (\text{unit } a) \ominus f\,a && \text{(because } f\,a \text{ evaluates to } v\text{)} \\
\equiv\ & (\text{unit } a) \ominus (\text{unit } a \gg\!= f) && \text{(by the } \textit{left identity} \text{ monad law)} \\
\equiv\ & (\text{unit } a) && \text{(by } \textit{strong left select} \text{ with } u = \text{unit } a\text{)}
\end{aligned}$$

- Showing that *left select* holds whenever *strong left select* holds is similarly simple:

$$\begin{aligned}
& u \ominus (\text{map } f\,u) \\
\equiv\ & u \ominus (u \gg\!= \text{unit} \circ f)\ u && \text{(expand the definition of } \textit{map}\text{)} \\
\equiv\ & u && \text{(by } \textit{strong left select} \text{ with } g = \text{unit} \circ f\text{)}
\end{aligned}$$

In the next section, we discuss commutative monads, which is another computation that is closely related to joinads.

5.2 Commutative monads

A monad is commutative if it obeys an additional *commutativity* law. Intuitively, the law specifies that the order of binding doesn't matter. When using monads to control effects, this means that the order in which the effects occur makes no difference. The law can be written as follows:

$$m \gg\!= \lambda v_1 \rightarrow n \gg\!= \lambda v_2 \rightarrow e \qquad \text{(commutativity)}$$
$$\equiv\ n \gg\!= \lambda v_2 \rightarrow m \gg\!= \lambda v_1 \rightarrow e$$

Many of the monads used in practice are commutative, including Reader (used for reading values from an environment), Maybe (representing computations that may fail) or Random (computations that use random numbers). In a retrospective on Haskell, Peyton-Jones considered working with commutative monads as an interesting open problem [12]. Although they are not sequential, the do-notation in Haskell allows only a sequential use.

Merge using commutative monads

Commutative monads are interestingly related to our work. They give us an implementation of the *merge* operation that obeys all required laws for free. We can implement merge in two ways and they give an equivalent operation thanks to the *commutativity* law. The two definitions look as follows:

$$\textbf{let}\ (\Vert)\ u\ v = \quad u \gg\!= \lambda a \rightarrow \quad v \gg\!= \lambda b \rightarrow \text{unit}\ (a, b)$$
$$\equiv$$
$$\textbf{let}\ (\Vert)\ u\ v = \quad v \gg\!= \lambda b \rightarrow \quad u \gg\!= \lambda a \rightarrow \text{unit}\ (a, b)$$

For commutative monads, the code above gives a valid definition of *merge*. A proof can be found in our previous work [13]. The commutativity law plays an essential role – for a monad that is not commutative, the two definitions mean different thing. For example, in the computation for working with events, the order of binding matters. Waiting for the first event and then for the second one is different than the reverse. If the second event occurs before the first one, we need to wait for the second occurrence (in the reversed order, we could run the continuation immediately after the first one).

Pattern matching syntax

When working with commutative monads just using bind and return, we're forced to use a sequential notation (as noted by Peyton-Jones [15]). Our monadic 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.

We have four values that represent a rectangle location (`mleft`, `mtop` and `mwidth`, `mheight`) and we want to calculate the center of the rectangle. If the computations couldn't fail, we could simply write:

```
(mleft + mwidth/2, mtop + mheight/2)
```

Unfortunately, inside monadic computation, we first need to extract the actual values using four bindings. In the F# syntax, this means using `let!` four times:

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

Since Maybe monad is commutative, we can mechanically define the *merge* operation in terms of *bind* and *return*. Then we can use the `match!` syntactic sugar to write:

```
maybe { match! mleft, mtop, mwidth, mheight 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 our work, we can see that our generalized pattern matching construct is certainly interesting from this point of view as well.

6. Future work and open problems

The goal of this document is to provide the simplest possible definition that is easy to explain and can be reasoned about in a formal manner. However, for practical purposes, we may want to make some of the primitive operations more powerful. Then we also briefly mention other computations that could benefit from the `match!` syntax.

6.1 Encoding of joinads

In this section, we discuss several alternatives for defining joinads. There are several aspects that are made implicit (by requiring the monad to support some feature) and that could be instead made explicit in the translation.

Explicit failure

In the definition of joinads described in this document, we require a special operation *fail* to encode failure of pattern matching inside the monadic computation. The laws then specify how should the *choose* operation behave with respect to *fail*.

An alternative approach would be to encode failure explicitly. We used this approach in our earlier work [5], but we had a more complicated definition of *choose*, so the situation is somewhat now. We could change the translation to use `option` type and translate clauses as follows (using example from §2.1):

```
input ≫= function 0 -> Some(unit 0) | _ -> None)
```

However, the *choose* operation is supposed to select the first non-failing computation, so it needs to be aware of failures. In the original translation, a clause is represented using a type `m (m a)` but choose only needs to be aware of the outer `m`, because a failure is represented in the outer computation (using *bind* with continuation containing *fail*). In the new translation, the type of clause would be `m (option (m a))` and the *choose* operation would have to have a type:

$$\ominus : \mathrm{m}\ (\mathrm{option}\ \mathrm{a}) \to \mathrm{m}\ (\mathrm{option}\ \mathrm{a}) \to \mathrm{m}\ (\mathrm{option}\ \mathrm{a})$$

Note that the resulting type also contains `option`, because we need to aggregate multiple clauses using the same operation. Depending on the computation, the operation may produce `None` value as it can be a suitable way to implement choosing. To remove the `option`, we would need a special operation of type `m (option a) -> m a`.

Intuitively, this operation encapsulates the "match failure" behavior. For example, for identity monad (extended to a joinad), it would throw match failure exception when the produced value were `None` (which is what ML languages do for standard `match`).

Explicit delay
In this document, we focused on the essence of joinad operations. To make joinads practically useful in eager languages (such as F#), we also need to consider side-effects. In particular, when we return a translated body of a computation from a translated clause, we don't want to run any side-effects that may be present in the body until the body is actually selected. F# computation expressions handle this problem and allow us to define delay operation with the following type:

delay : (unit → m a) → m a

The operation takes a function that may have side-effects and wraps the side-effects inside a monad. This means that the function can be defined only for monads that are delayed (e.g. continuation, but not Maybe). The call to *delay* is automatically inserted in places where side-effects may occur. We could use the operation to delay side-effects in the body of clause. Using the same example as in previous section:

```
input ≫= function 0 -> unit(delay (fun () -> unit 0)) | _ -> None)
```

If we require the monad to provide *delay*, we place a considerable number of demands on the author of the computation type (it must support *delay*, *fail* and *choose* shouldn't evaluate side-effects when checking whether a computation has failed). An alternative may be to delay bodies automatically and generate code like the follows:

```
input ≫= function 0 -> unit(fun () -> unit 0)) | _ -> fail)
```

To actually run the computation, we don't need any additional operation (unlike when representing failure explicitly). The resulting type of clause would be `m (m (unit -> a))`. We could force the evaluation when flattening the type by replacing "≫= id" at the end of the translated code with continuation that uses bind to extract the function and then runs it.

Efficient compilation
In our current compilation scheme, the values are merged before the pattern matching is performed. This means that we cannot tell whether a value available from one computation matches a pattern specified for that computation. We can only tell whether a combination of values matches a combination of patterns. This makes it possible to support the `when` clause in the compilation. Although this is not handled in the formal translation in §3, we implemented the feature in our prototype and used it in §2.2.

For some computations, it would be beneficial if we could test individual values against patterns independently. Qin Ma and Maranget show [14] how to implement such pattern matching for Join calculus. In Join calculus, this is very important as it allows efficient implementation. When a value becomes available, they don't need to try running the pattern matching for every combination. They just need to test the individual value and then can decide whether there is a suitable combination or not. Indeed, their translation cannot handle `when` clauses, which can specify a relation between the individual values. Modifying the translation of joinads to efficiently support Join patterns is an interesting open problem.

Supporting empty patterns
The translation presented in section §3.2 does not handle the case when a clause contains only ignore patterns. For example, the following code (using the time-varying computation) is not supported:

```
changing { match! c1, c2 with
           | !v1, !v2 -> return 1
           | _, _ -> return 0 }
```

What would be the meaning of this computation? Intuitively, it would be logical that the computation creates a time-varying value that is 1 when both of the source computations are defined and 0 otherwise. For `changing` computation, we could get this behavior using *unit*.

However, using *unit* doesn't make sense for other computations. When working with futures or events, the last clause would be executed always, even before waiting until any of the events or futures produce a value. This seems to be clearly useless. For futures, we may want to run the clause only when all futures complete, but if there is no other matching clause, but that can be written using the `!_, !_` syntax. We are not convinced whether supporting empty clauses would be a useful addition to the translation and how should it be done to provide a reasonable meaning for all computations.

6.2 Other computation types

The definition of joinads extends monads and we used it to encode computations for working with futures, time-varying values and events. However, it may be also possible to define joinads for other abstract computations such as Arrows [15] or Idioms [16]. We'll discuss the latter option briefly in the next section. Another interesting problem is using joinads for encoding other programming models.

Idioms and joinads

Idioms (also called *applicative functors*) [16] are abstract computations that are weaker than monads. This means that every monad defines an idiom, but not all idioms are also monads. Consequently, a joinad also defines an idiom. There are various equivalent definitions of idioms. The one that shares the most similarities with monads and joinads requires the following three operations:

$$
\begin{aligned}
&\mathit{unit} &&: \mathrm{a} \to \mathrm{f\ a} \\
&\mathit{map} &&: (\mathrm{a} \to \mathrm{b}) \to \mathrm{f\ a} \to \mathrm{f\ b} \\
&\star &&: \mathrm{f\ a} \to \mathrm{f\ b} \to \mathrm{f\ (a, b)}
\end{aligned}
$$

The signature of the $\star$ operation should look very familiar. The type is the same as the type of the *merge* (⩔) operation of joinads. Even though the types are the same, idioms and joinads require different sets of laws. Idioms require the following:

$$
\begin{aligned}
\mathrm{map\ assoc}\ (u \star (v \star w)) &\equiv (u \star v) \star w && \text{(associativity)} \\
\mathrm{map}\ (f \times g)\ (u \star v) &\equiv \mathrm{map}\ f\ u \star \mathrm{map}\ g\ v && \text{(naturality)} \\
\mathrm{map\ snd}\ (\mathrm{unit}\ a \star v) &\equiv v && \text{(left identity)} \\
\mathrm{map\ fst}\ (u \star \mathrm{unit}\ a) &\equiv u && \text{(right identity)}
\end{aligned}
$$

The first two laws – *associativity* and *naturality* – are required for both $\star$ and for our ⩔ operation. The next two (identity) laws are required only for $\star$ and do not have to hold for joinads. For example, the definition of ⩔ for events does not obey them. On the other hand, joinads are required to obey the commutativity law, which is not required by idioms. However, there are some *commutative idioms* that obey the law. Examples include any instances derived from commutative monads such as IO, Reader and Maybe, but also computations that are only idioms such as ZipList.

As already mentioned, every monad and therefore also joinad is an idiom. The $\star$ operation can be implemented in terms of *bind* and *return*. However, for some joinads, we may give an alternative definition by using ⩔ as a definition of $\star$. This is not possible automatically as the laws differ, but if we verify that left and right identity holds, we may get an alternative idiom.

Pattern matching for Idioms

Another interesting question related to idioms is whether we could desugar the `match!` construct using the operations provided by idioms, but without the *bind* operation that is available only in monads. This could make the syntax useable for a wider range of computations.

The operation ⋆ appears like a good candidate for combining multiple input computations into a single one (instead of ⓫). To guarantee that reordering doesn't change the meaning, we would likely need to add the requirement for commutativity.

Implementing choosing between clauses is a more interesting problem. Using ⊜, we can select between two computations, but idioms do not give us any way to flatten a computation of type `M<M<T>>`. We'd need to add some additional operation that allows us to choose between several options. This would make idioms more expressive, as we could write computations whose structure depends on the values. This is not possible for pure idioms as discussed in [16], section 5. However, such computation could still be weaker than monads. The structure could only select between several options, but could not be completely arbitrary as with monads.

Encoding Join calculus

Join calculus [19] is a programming model for writing concurrent applications. A program consists of *channels* and *joins*. A join specifies that some operation should be executed when there are values in certain channels. The operation may run any imperative commands or it can send messages to channels.

In our previous work [5] we used join calculus as one of the computations that can be encoded using joinads. However, the programming model may not fit our new definition of joinads, because it is not clear how to define join calculus using monads. In our earlier attempt, the body of `match!` clauses could be imperative computations and we didn't need to provide *unit* operation. The body was just executed (e.g. to send messages to other channels) and produced no return value.

Alternatively, we could define a computation for working with values of type `Channel<T>`. Using `match!` we could create channels that are defined in terms of values available in other channels. For example, to implement a concurrent buffer with two input channels, we could write something like:

```
let putInt = new Channel<int>()
let putString = new Channel<string>()
let get = new Channel<ReplyChannel<string>>()

let buffer : Channel<JoinOperation> = join {
  match! get, putInt, putString with
  | !chnl, !n, _ -> return Reply(chnl, "Number: " + (string n))
  | !chnl, _, !s -> return Reply(chnl, "String:" + s) }

buffer.Start()
```

The snippet first declares three channels (the `get` channel carries reply channels, which are essentially continuations to be called when a value in the buffer becomes available). Then we use `match!` to define a channel `buffer` that carries operations specifying what should be done when a combination of values is available. The `Start` operation (available for `Channel<JoinOperation>` only) would then perform the join calculus operations as they appear in the channel.

One problem with this encoding is that it is not entirely clear how should the *bind* operation for channels behave. This suggests that join calculus may be better encoded using `match!` construct built on top of idioms. It should be possible to provide standard operations for idioms (*map*, *unit* and ⋆). With an appropriate extension to support choosing between computations, idioms may give us a way to encode joins as well.

7 Related work and conclusions

In this document, we extended monadic computations and the F# syntax for working with them to support 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 [18], there are several other abstract types of computations. We already mentioned idioms, but it could be interesting to consider whether generalized pattern matching would be a useful extension for the other:

- Idioms (or applicative functors) [16] provide an abstraction that is more common than monads, but less powerful. As already mentioned in section §6.2, we believe that adding support for pattern matching and choice to idioms could be an interesting future direction.
- Co-monads (a categorical dual of monads) [20] have shown useful for data-flow programming [22]. As far as we're aware, there is currently no syntactic extension (akin to the do-notation) for co-monads and so it is difficult to imagine how co-monadic computations could benefit from pattern matching support.
- Arrows [15, 7] are used mainly in functional reactive programming research, which is based on working with time-varying values and discrete events. Arrow computations can be written using the arrow notation [24] and generalized pattern matching may be an interesting extension.

Pattern matching
The work on pattern matching mainly focused on providing better abstraction when pattern matching on normal values [25, 26]. Extensible pattern matching in Scala [29] 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 document.

Using MonadPlus type class
Some authors proposed to generalize the return type of a pattern from `Maybe` to any instance of Haskell's `MonadPlus` type class [27, 28]. To some point, this is similar to our proposal (as mentioned in the introduction, types of our *choose* and *fail* match the type class). However, we believe that a different set of laws is essential for understanding the computation. For example, we could define a pattern `Id` that returns the list given as an argument and then write (using syntax from [28]):

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

The proposed desugaring uses the Haskell's `MonadPuls` type class is shown below. For lists, the `mplus` operation stands for list concatenation and `mzero` creates an empty list:

```
mplus
  (Id [0; 1] >>= fun x -> Id [2; 3] >>= fun y ->
      if x = 0 then unit y else mzero)
  (Id [0; 1] >>= fun x -> unit x)
  (Id [0; 1] >>= fun x -> unit x)
```

If we executed the example, the result would be [2; 3; 0; 1; 0; 1]. This result may be a surprise for some users. When using the `MonadPlus` type class, the computation executes all clauses for which the pattern matching succeeds and then combines the results. For lists, this means that the resulting list combines values generated by all clauses.

The encoding using `MonadPlus` doesn't respect some of the equations that we expected. Most importantly, it breaks the *Match first* equation. For monads that are not commutative, it would also break the *Reordering* equation. We find this counterintuitive and would instead expect the pattern matching to use zip-semantics for lists. Using our proposal, we can get this behavior if we with computations of type `list<option<T>>` and use `None` to represent failure.

For some types that do not represent computations in the usual sense (such as parsers), breaking the *Match first* equation may be the right thing to do. However, we believe that for most of the computations, this (and other) equations are important to preserve the usual intuition about the construct that users already have.

Applications

In this document, we used our language extension to encode several programming models that were introduced in the academic literature. We presented a parallel programming model based on futures [17]. The `match!` construct defined by this model is very similar to the `pcase` (parallel case) construct introduced in Manticore [12].

The next two examples (time-varying values and events) come from the Functional Reactive Programming (FRP) research that was first described by Elliott and Hudak [4]. Our time-varying values are very similar to *behaviors* used in FRP. Our computation for working with events is more closely related to the event-based aspects of FRP [30]. We almost directly implement the semantics of *bind* operation that is defined in imperative streams [2], although we're working in an impure language, so we do not embed side-effects inside the monad.

In the future work section, we mentioned that it may be possible to implement an extension similar to ours for *idioms* as well. We believe this would give us a way to implement programming model based on join calculus [19]. This can be already done using extensible pattern matching in Scala [8]. However, our approach could benefit from asynchronous workflows [23]. This would allow 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).

7.1 Conclusions

In this document, we discussed a definition of joinad computation that requires all joinads to also be monads. In addition to the usual monad operations, joinads need to provide three additional operations. *Fail* represents a pattern matching failure, *merge* ⫫ is used to aggregate patterns of `match!` clauses vertically and *choose* ⊖ is used to aggregate clauses horizontally.

The three operations have a simple type signature and we discuss laws that should hold about them in order to preserve the usual intuition about ML-style pattern matching. The laws we describe are motivated by refactorings that shouldn't change meaning of pattern matching. The fact that similar laws often appear in related computation types suggests that our choice of laws is correct. We discussed how our laws related to similar computation types. Most importantly, we support the proposal to define a `MonadOr` type class for Haskell libraries, but we suggested a slight refinement of the required laws in order to include joinad computations defined by *choose* and *fail*.

We used three computations to demonstrate joinads. The computations are *futures*, *time-varying values* and *events*. All of them are directly inspired by existing computations described in the literature. To precisely specify how the three examples work, we described the semantics of joinad operations for these computations.

References

[1] MonadPlus reform proposal - HaskellWiki. Available online at: http://www.haskell.org/haskellwiki/MonadPlus_reform_proposal

[2] E. Scholz: Imperative streams - a monadic combinator library for synchronous programming. In Proceedings of ICFP 1998

[3] M. Fluet, M. Rainey, J. Reppy, A. Shaw: Implicitly-threaded parallelism in Manticore. In Proceedings of ICFP 2008

[4] C. Elliott and P. Hudak, Functional reactive animation. In Proceedings of ICFP 1997

[5] T. Petricek, D. Syme: Joinads: A Retargetable Control-Flow Construct for Reactive, Parallel and Concurrent Programming. In Proceedings of PADL 2011

[6] D. Syme: F# Language Specification. http://tinyurl.com/fsspec

[7] H. Liu, E. Cheng, P. Hudak: Causal commutative arrows and their optimization. In Proceedings of ICFP 2009

[8] P. Haller, T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. In Proceedings of COORDINATION 2008

[9] MonadPlus – HaskellWiki. Available online at: http://www.haskell.org/haskellwiki/MonadPlus

[10] Control.Monad – GHC Documentation. Available online at: http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html

[11] J. Voigtländer: Free theorems involving type constructor classes. In Proceedings of ICFP 2009

[12] S. Peyton Jones. Wearing the hair shirt - A retrospective on Haskell. Invited talk, POPL 2003. Slides available online at: http://tinyurl.com/haskellretro

[13] T. Petricek: Reactive Programming with Events (Master thesis) Charles University, 2010

[14] Ma Qin, L. Maranget. Compiling Pattern-Matching in Join-Patterns, In Proceedings of CONCUR 2004

[15] J. Hughes, Generalising Monads to Arrows, In Science of Computer Programming 37, pp67-111, May 2000.

[16] C. McBride and R. Paterson, Applicative programming with effects Journal of Functional Programming 18 (2008)

[17] H. Baker, C. Hewitt. "The Incremental Garbage Collection of Processes". In Proc. Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12.

[18] P. Wadler. Monads for functional programming. In Advanced Functional Programming, LNCS 925, 1995.

[19] C. Fournet, G. Gonthier: The reflexive CHAM and the join-calculus. In Proceedings of POPL 1996.

[20] R. Kieburtz. Codata and Comonads in Haskell. Unpublished draft, 1999. Available online at: http://tinyurl.com/comonads

[21] M. Tullsen. First class patterns. In Proc. PADL, 2000

[22] T. Uustalu, V. Vene. The essence of dataflow programming. In Proceedings of APLAS 2005

[23] D. Syme, A. Granicz, and A. Cisternino. Expert F#, Reactive, Asynchronous and Concurrent Programming. Apress, 2007.

[24] R. Paterson. A new notation for arrows. In Proceedings of ICFP 2001

[25] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Proceedings of POPL, 1987

[26] C. Okasaki. Views for Standard ML.
In Proceedings of Workshop on ML, Baltimore, Maryland, USA, pp. 14–23, 1998.

[27] M. Tullsen. First class patterns. In Proc. PADL, 2000

[28] D. Syme, G. Neverov, J. Margetson. Extensible Pattern Matching via a Lightweight Language Extension. In Proceedings of ICFP 2007.

[29] B. Emir, M. Odersky, J. Williams: Matching Objects with Patterns. In ECOOP 2007.

[30] C. Elliott: Declarative event-oriented programming. In Proceedings of PPDP 2000