Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming

Tomas Petricek[1], Don Syme[2]

[1] Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic
[2] Microsoft Research, Cambridge, United Kingdom
`tomas@tomasp.net, dsyme@microsoft.com`

Abstract. Modern challenges led to a design of a wide range of programming models for reactive, parallel and concurrent programming, but these are often difficult to encode in general purpose languages. We present an abstract type of computations called *joinads* together with a syntactic language extension that aims to make it easier to use joinads in modern functional languages.

Our extension generalizes pattern matching to work on abstract computations. It keeps a familiar syntax and semantics of pattern matching making it easy to reason about code, even in a non-standard programming model. We demonstrate our extension using three important programming models – a reactive model based on events; a concurrent model based on join calculus and a parallel model using futures. All three models are implemented as libraries that benefit from our syntactic extension. This makes them easier to use and also opens space for exploring new useful programming models.

1 Introduction

Today, we often write programs for environments that are in some way non-standard when contrasted to traditional expression-based computation. In parallel programming, multiple functions can execute at one time; in concurrent programming, we need to express synchronization of multiple processes; in reactive programming, we write code that waits for events from the GUI or completion of background tasks and acts in response. Academia offers many programming models for these domains, and more and more of them are being used by main-stream developers, though often awkwardly through object-model, library-based encodings.

This raises the question of providing language support for those models. Specialized languages become overly specific, while library-based solutions often result in unnatural encodings where the declarative intent of the program is lost. We believe that the best option lies in between. If we identify a repeating pattern, we can provide a syntactic extension that enables a large number of programming models. This approach is successfully utilized by Haskell's monads [2], computation expressions in F# [1] and LINQ queries in C# [25]. Language supported, pattern-based approaches are particularly appealing in the area of reactive, parallel and concurrent programming, where we need to choose between different programming models.

In this paper, we identify a repeating pattern that we call *joinad*. It arises when we need to pattern match on abstract computations as opposed to pattern matching on concrete values. The key contributions of our work are the following:

Practically useful. Joinads naturally fit with many important programming models. Section 2 supports this claim by showing a reactive programming model (Section 2.1) inspired by imperative streams and FRP [17, 23]; a concurrent programming model (Section 2.2) based on join calculus [5] bearing similarities to JoCaml and Cω [6, 7]; and a parallel programming model (Section 2.3) based on futures, which can nicely express some aspects of Manticore [12].

Lightweight extension. We present a construct that allows pattern matching on abstract computations (e.g. event, channel or future). The construct is just a syntactic sugar and is translated into calls to two simple operations provided by a joinad. We describe the two operations as well as the translation procedure (Section 3).

Well-founded. As usual when describing abstract computation types, we identify a set of laws that needs to be followed by joinad operations. We chose laws such that our generalized pattern matching construct keeps the familiar semantics of ML-style pattern matching (Section 4) and we describe the relationship between joinads and other abstract computations (Section 5), most notably commutative monads.

This paper presents joinads as an extension to F# computation expressions. Thanks to their relations with monads, the presented ideas could be applied to any language with support for monads. We start by giving background on F# computation expressions.

1.1 Computation expressions

Computation expressions [1, 3] are a syntactic mechanism in F# that provides convenient syntax for a range of computations. As with Haskell monadic syntax and LINQ queries, F# computation expressions are just a syntactic mechanism. In practice, they are usually used with established computation type (e.g. monoids, monads or additive monads [4; Ch. 2]) which satisfies specific laws.

We demonstrate computation expressions using a reactive programming model described in detail in [4]. As we'll see later, the work presented in this paper can be used (among other things) to encode complex interaction patterns in this reactive programming model. We work with values of type `Event<'T>`, which represents running computations that emit values of type `'T` along the way. The type can be modeled as a sequence of time-value pairs. The following example shows a counter of button clicks that limits the rate of clicks to one per second:

```
1: let rec counter n = event {
2:   let! me = btn.Click
3:   let! _ = Event.sleep 1000
4:   return n + 1
5:   return! counter (n + 1) }
```

Let's look what the code does assuming appropriate definitions of `event`, `Event.sleep` and the `Click` property. The recursive function returns a computation `Event<int>`. Its body is wrapped in an `event { ... }` block, which provides the meaning of constructs such as `return`, `return!` and `let!` The computation starts by waiting for the `btn.Click` event (line 2). The meaning of the `let!` construct is that it waits for the first occurrence of the specified event and runs the rest of the code once afterwards. Next, we create an event that will occur after 1 second and wait for its occurrence (line 3).

The `return` construct is used to emit values from the event (line 4). We can call it multiple times because an event may be triggered repeatedly. The `return!` construct performs a tail-call to implement looping and wait for the next `Click`.

In computation expressions, the semantics of the control-flow in the syntactic fragment enclosed by `event { .. }` is determined by the operations on the `event` value. The expected types of operations and translation rules are defined in [3], and in this case the `event` value supports the following operations:

```
event.Bind       : M<'T> → ('T → M<'R>) → M<'R>
event.Combine    : M<'T> * M<'T> → M<'T>
event.Return     : 'T → M<'T>
event.ReturnFrom : M<'T> → M<'T>
```

The type signatures bare similarity to the `MonadPlus` typeclass in Haskell, although the library for events described above does not satisfy the usual `MonadPlus` laws. The following snippet demonstrates how the translation looks for the above example.

```
let rec counter n =
  event.Bind (btn.Click, fun me ->
    event.Bind (Event.sleep 1000, fun _ ->
      event.Combine (
          event.Return n,
          event.ReturnFrom (counter (n + 1)))))
```

Uses of the `let!` construct are translated into calls to the `Bind` operation and the rest of the computation is transformed to a continuation. In this example, binding waits for the first occurrence of an event, and so the continuation will be called at most once, but other computations may run it multiple times (e.g. each time an event occurs).

The `return` construct is translated into calls to the `Return` operation, which has the same type signature as monadic *unit* and lifts a value `'T` into a computation `M<'T>`. The `return!` construct translates to the `ReturnFrom` operation, in this case implemented as an identity function. Finally, when we sequence multiple event generators, the computations are combined using the `Combine` construct.

2 Joinads by example

In this section, we introduce our lightweight syntactic extension and we'll explore several practically useful programming models that can benefit from it. The translation to underlying operations will be discussed later in section 3.

2.1 Reactive programming with Events

First we show a more complicated example of user interaction logic using the reactive programming model from the previous section. Let's say that we want to reset the counter by pressing the `Esc` key. In practice, this means that we need to wait for either `Click` event or `KeyDown` event that carries the `Esc` key code as a value. Unfortunately, this cannot be written directly using existing constructs. Using `let!` we can wait for multiple events only sequentially, but not in parallel.

What do we do about this? One approach is to use a combinator library that allows us to filter and compose events. However, a combinator approach to waiting for multiple events makes the syntax more involved and forces the programmer to leave the computation expression syntax. A solution using this approach is available in Appendix A [27] for a comparison. The alternative approach described in this paper is to add a new syntactic control flow construct to computation expressions to express joining computations. What should this control flow operator look like? It should

- accept multiple computations as inputs,
- select a computation path based on the values produced by computations, and
- enable its use with different computation types (be retargetable).

In functional languages, the similarity to pattern matching is easy to note. In ML-like languages, the `match` construct accepts multiple values as inputs, and selects a computation path based on the inputs. In our proposal, the `match!` construct plays an analogous role for computations. Similarly, just as `let!` allows binding on computation values, `match!` allows pattern matching on computation values. The resettable counter can be written as follows:

```
1: let rec counter n = event {
2:   match! btn.Click, win.KeyDown with
3:   | !_, _   -> let! _ = Event.sleep 1000
4:                return n + 1
5:                return! counter (n + 1)
6:   | _, !Esc -> return! counter 0 }
```

The `match!` construct takes one or more computations as arguments (line 2). In our example, we give it two values of type `Event<'T>`. The patterns (lines 3, 6) belong to a syntactic category that we call *computation patterns*. The form "!<*pat*>" means that we need to obtain a matching value from the computation (in case of events, we wait until the event emits a value matching the underlying ML-style pattern <*pat*>). We call this form a *binding pattern*. The second form (written as "_") is called *ignore pattern*. It specifies that we don't need to obtain any value from the computation. Note that there is a difference between "_" and "!_" (line 3). 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 we ignore it afterwards.

The meaning of `match!` in the event-based reactive programming model is that it waits for the first combination of event occurrences that enables a particular clause (when waiting for multiple events, the values of last occurrences are remembered). In the previous example, each clause has only a single *binding pattern* meaning that each clause waits only for a single event. In the second clause (line 7), the value has to match the pattern `Esc`, so some occurrences of the `KeyDown` event will be ignored.

As we'll see in section 4, `match!` should generalize the `let!` construct. This is indeed the case for events – if we pattern match only on a single computation and specify an irrefutable pattern, the behavior is the same as when using `let!`

2.2 Concurrent programming with Joins

Our second example 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, 10].

Programming model based on Join calculus expresses synchronization using *channels* and *join patterns*. A channel can be viewed as a thread-safe container into which we can put values without blocking the caller. A join pattern is a rule saying that a certain combination of values in channels should trigger a specified reaction (and remove values from the channels). We can use `match!` to specify the combinations of values by pattern matching on multiple channels of type `Channel<'T>`. A simple unbounded buffer can be implemented as follows:

```
1: let put = new Channel<int>()
2: let get = new Channel<ReplyChannel<int>>()
3:
4: let buffer = join {
5:   match! put, get with
6:   | !num, !chnl -> chnl.Reply num }
```

We start by defining two channels (lines 1, 2). The first one is used for putting values into the buffer, and the second one for obtaining them. The type `ReplyChannel<int>` is essentially a continuation taking `int`. In our example, the continuation will be invoked by the buffer as soon as a value (provided by a call to `put`) is available.

The buffer is implemented using the `match!` construct provided by the `join` computation expression. Join patterns are encoded as clauses of `match!` In our example, we have a single clause (line 6) consisting of two bindings. This means that the body will be called when there is a value in the `put` channel and also a continuation in the `get` channel. When the join pattern fires, we pass the `num` value to the continuation.

The match! construct becomes essential when we have multiple join patterns, each of them binding on one or more channels. The next example shows a buffer that allows storing of two distinct types of values using two input channels. Values can be read using a `get` channel that returns them as strings. This logic can be encoded using two join patterns that bind on the `get` channel and one (`putInt`) or the other (`putString`) channel for storing values:

```
1: let putInt = new Channel<int>()
2: let putString = new Channel<string>()
3: let get = new Channel<ReplyChannel<string>>()
```

```
4: let buffer = join {
5:   match! get, putInt, putString with
6:   | !chnl, !n, _ -> chnl.Reply ("Number: " + (string n))
7:   | !chnl, _, !s -> chnl.Reply ("String:" + s) }
```

Each clause combines two channels (lines 6 and 7) and ignores the third one. If we get an integer value and a reply channel `chnl` in the first join pattern (line 6), we send a number converted to a string as the reply. The second clause is quite similar.

2.3 Parallel programming with Futures

The next example shows how to multiply values in a binary tree. We use futures – values of type `Future<'T>` that represent a computation that is (or may be) running in the background and eventually produces a value of type `'T`. A computation `future` creates a future and can wait for the results of another future using `let!` The `match!` extension allows us to wait for multiple features and pattern matches on the results:

```
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 creates a future. It starts by standard pattern matching on the tree (line 2), which is just a discriminated union. If the tree is a node, we recursively call the `treeProd` function to create two futures to process both of the branches (line 4). Then we need to wait for both of the futures to produce a value, which is done using pattern matching on computations with two binding patterns (line 7). In case when one future completes earlier and produces 0, we know the overall result immediately, and we can return it (lines 5 and 6) and the computation automatically cancels remaining futures.

When using `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 remaining futures. In the other case, it waits for more futures to complete. This behavior is in many ways similar to the `pcase` construct in Manticore [12].

3 A language extension for joinads

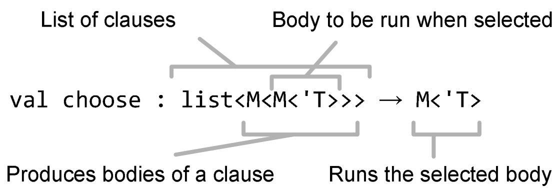
In this section, we present our language extension for F# in detail. Just like other aspects of F# computation expressions, it is a retargetable control flow construct implemented by a syntactic translation to function calls. We first show how the translation works on the examples from the previous section and then present formal translation rules. The joinad operations and laws are discussed in section 4.

3.1 Introducing operations

The translation of `match!` requires three functions – the usual *map* operation and two additional operations that we call *merge* and *choose*. In this section, we gradually introduce how the translation works, starting with a case where we need only *map* and a slightly simplified *choose* that doesn't allow refutable patterns in `match!` clauses.

Simplified choose. We start by looking at the example from section 2.1, but we ignore the fact that the second clause contains a pattern that may fail – we reset the counter whenever `KeyDown` occurs. This way, we get an example with multiple clauses where each clause contains a single binding with an irrefutable pattern.

In this case, we only need an operation that allows us to select one of the clauses. This is the purpose of the *choose* operation, which is explained in figure 1. The translation also needs the *map* operation, which allows us to transform values "inside the computation" and has the usual type `('T → 'R) → M<'T> → M<'R>`.



```
val choose : list<M<M<'T>>> → M<'T>
```

Fig. 1. The *choose* operation takes a list of computations. Each of the computations in the list carries (or produces) other computations. These wrapped computations represent the body of the clause that should be called when the clause is selected.

If you're familiar with the definition of monads in terms of *join*, *map* and *unit*, you probably noticed that our *choose* operation looks similar to *join*, except that it takes a list of `M<M<'a>` computations instead of just a single one. As we'll see in section 4, when a joinad is also a monad, *choose* should be a generalization of *join*. The following code shows desugared version of the example from section 2.1:

```
1: let rec counter n =
2:   choose [
3:     map (fun me -> event {
4:       let! _ = Event.sleep 1000
5:       return n + 1
6:       return! counter (n + 1) }) btn.Click;
7:     map (fun ke -> event {
8:       return! counter 0 }) win.KeyDown ]
```

The two clauses are translated into two elements of a list passed as the argument to *choose* (line 2). Each computation representing a clause is constructed by taking the source event and projecting values emitted by the event into `event` computations representing the body that should be executed when the clause is selected. This is done using the *map* operation (lines 3 and 7).

Merge. In the previous example each clause contained only a single binding pattern. This means that we didn't need to obtain values from a combination of computations. If we wanted to do that, we would need some way of merging two computations into a single one carrying tuples. To enable this, we need a *merge* operation with the following type signature:

```
val ⦷ : M<'T> → M<'U> → M<'T * 'U>
```

The merge operation takes two computations that may produce value of different types and constructs a single computation that produces a tuple of values. The meaning of the operation depends on the computation, but as we'll see in section 4, it should obey certain laws. We'll discuss how the operation relates to monads in section 5 and focus on the translation for now. The following example shows a translated version of the first join pattern example from section 2.2:

```
1: let buffer =
2:   choose [ map (fun (num, chnl) ->
3:     join { chnl.Reply num }) (put ⦷ get) ]
```

The example uses only a single clause, so the list passed to *choose* consists of a single element. However, the clause binds on multiple channels, so we need to obtain values from both of the channels simultaneously. This is achieved by merging the channels using the ⦷ operator (line 3) and then passing the merged channel as an argument to the *map* operation.

The implementation of the *merge* operation for join channels is perhaps the most complicated of the three examples presented in this paper. It creates a new channel, but when a clause is selected in *choose*, we need to remove values from the original channels (e.g. `put` and `get`). This can be done by creating an *alias channel* that keeps reference to the two merged channels.

Choose with failures. Earlier we wrote that *choose* takes a list of computations that contain computations to be used if the clause is selected. This simplification does not take failure into account. The outer computation consists of pattern matching that may fail or succeed. In the second case, it produces an inner computation that can be used to continue with. As a result, the actual type signature of *choose* is the following[1]:

```
val choose : list<M<Option<M<'T>>>> -> M<'T>
```

When compared with the signature shown earlier, the only change is that the inner computation of type `M<'T>` is now wrapped in the `Option<'T>` type. This allows us to represent pattern matching failure using the `None` case.

We show the handling of patterns by looking at the translation of an example from section 2.3, which used futures to multiply leaves of a tree. The next snippet shows the code generated for the last two clauses of the example (one that returns 0 when the

[1] In a strict language like F#, we also need to delay the inner `M<'T>` value to ensure that its side-effects are evaluated only when a clause is actually selected. We omit this detail for simplicity.

second future yields 0 and the general case where we wait for both of the futures). The values `f1` and `f2` store the result of calling `treeProd` on `lt` and `rt` respectively:

```
1: choose [
2:   ...
3:   map (function
4:     | 0 -> Some(future { return 0 })
5:     | _ -> None) f2;
6:   map (function
7:     | a, b -> Some(future {
8:         return a * b })) (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` will be called with a value produced by the future. If the value is 0, it returns `Some` with a future computation to run (line 4) otherwise it returns `None` (line 5). The second clause is similar, with the exception that it first combines two futures using the ⦷ operator. Also, the pattern matching always succeeds, so we can omit the `None` case.

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 `Some`, the `choose` operation cancels all other futures in the list (which in turn cancels the `f1` future) and runs the body provided in the `Some` discriminator. In case of non-zero result, it continues waiting until some other clause produces `Some`. If all clauses produce `None`, then the `choose` operation throws a match failure exception.

3.2 Syntax extension

Let's now look at the syntax of the extension. In addition the standard constructs described in [3], 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 computation clauses.

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

Clauses do not consist of standard patterns, but are formed by computation patterns. As a result, 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 [3] prefixed with "!". In the next section, we describe a translation that transforms computation expressions that include `match!` into ordinary F# expressions.

3.3 Translation semantics

We extend the translation defined in the F# specification [3] by adding a case for the `match!` construct. 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:

$$
\begin{array}{lcl}
[\![-]\!] & : & \mathrm{expr} \rightarrow \mathrm{expr} \\
\langle\!\langle - \rangle\!\rangle & : & \mathrm{cexpr} \rightarrow \mathrm{ident} \rightarrow \mathrm{expr} \\
\langle - \rangle & : & \mathrm{ccl} \rightarrow \mathrm{ident} \times [\,\mathrm{ident}\,] \rightarrow \mathrm{expr}
\end{array}
$$

In section 1.1, we saw that 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 $merge_m$ to denote the *merge* operation provided by the builder *m*. When translating a clause, we also need the parameters of `match!` These are stored in fresh values and passed to the function as a list of identifiers:

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

$$
\begin{array}{l}
\langle\!\langle\, \mathbf{match!}\ expr_1, \ldots, expr_k\ \mathbf{with}\ ccl_1 \mid \ldots\ ccl_p\, \rangle\!\rangle_m \quad \equiv \\
\quad \mathbf{let}\ v_1 = expr_1\ \mathbf{in} \ldots\ \mathbf{let}\ v_k = expr_k\ \mathbf{in} \\
\quad \mathrm{choose_m}\ [\ \langle\, ccl_1\, \rangle_{m,\,(v1,\,\ldots,\,vk)};\ \ldots\ ;\ \ \langle\, ccl_p\, \rangle_{m,\,(v1,\,\ldots,\,vk)}\ ]
\end{array}
\tag{1}
$$

$$
\begin{array}{l}
\langle\, cpat_1, \ldots, cpat_k\ \texttt{->}\ cexpr\, \rangle_{m,\,(v1,\,\ldots,\,vk)} \quad \equiv \\
\quad \mathrm{map_m}\ (\mathbf{function}\ (pat_1, \ldots), pat_n \rightarrow \mathrm{Some}\ \langle\!\langle\, cexpr\, \rangle\!\rangle_m \\
\qquad\qquad\qquad \mid \_ \rightarrow \mathrm{None})\ cargs
\end{array}
\tag{2}
$$

$$\mathbf{where}\quad \{\ (pat_1, v_1), \ldots, (pat_n, v_n)\ \} = \{\ (pat, v_i) \mid cpat_i = !pat;\ 1 \leq \mathrm{i} \leq \mathrm{k}\} \tag{3}$$

$$cargs = v_1 \circledparallel_{\mathrm{m}} \ldots \circledparallel_{\mathrm{m}} v_{n-1} \circledparallel_{\mathrm{m}} v_n \quad \text{for}\ \ n \geq 1 \tag{4}$$

When translating `match!` (1), we construct a fresh value for each of the arguments. This guarantees that any side-effects of an expression used as an argument will be executed only once. The rest of the rule translates all clauses of the pattern matching and creates an expression that chooses one clause using the $choose_m$ operation.

When translating a clause (2), we need to identify which of the arguments are matched against a *binding pattern*. This is done in (3) where we construct a list containing an ordinary pattern (extracted from the binding pattern) and a computation, to be matched against it. Next we combine all needed computations into a single value using the merge operator (4). The operator is left-associative, so when combining for example three values, the resulting value will be of type `M<('a * 'b) * 'c>`.

Finally, we pass the combined computation as an argument to a map_m operation. In the projection function, we match the actual value against the patterns extracted earlier. If the matching succeeds we return `Some` containing a delayed and translated body of the clause. The result of translating a computation clause will be of a type `M<Option<M<'T>>>`.

4 Reasoning about joinads

So far we described the types of operations that a joinad defines and a translation of our lightweight language extension. Since joinad is an abstract type, we cannot specify the semantics of its operations in general. However, we can specify that they should follow certain algebraic laws. In this paper, we identify some of the laws that we would expect to hold about joinad operations. We do not claim a completeness result for these laws (c.f. Haskell Arrows [22, 26] where equations have been identified, but a completeness result is elusive).

When using standard pattern matching, we have an intuition about transformations that do not change the meaning of program. Since our `match!` construct bears a close resemblance to an ordinary `match`, we want to be able to perform similar syntactic transformations without affecting the semantics:

match! $m_{p(1)}$, … , $\mathrm{m}_{\mathrm{p(n)}}$ **with**
| $cpat_{1,\,p(1)}$, … , $cpat_{1,\,p(n)} \rightarrow cexpr_1$ | …
| $cpat_{k,\,p(1)}$, … , $cpat_{k,\,p(n)} \rightarrow cexpr_k$

…are equivalent for any permutation *p* of *n* numbers (1)

match! *m* **with**
| !var_1 -> $cexpr_1$
| !var_2 -> $cexpr_2$
≡
match! *m* **with**
| !var_1 -> $cexpr_1$ (2)

match! *m* **with**
| !$var \rightarrow cexpr$
≡
let! $var = m$
$cexpr$ (3)

match! m {**return** e_1},
m {**return** e_2} **with**
| !var_1, !$var_2 \rightarrow cexpr$
≡
m { **match** e_1, e_2 **with**
| var_1, $var_2 \rightarrow cexpr$ } (4)

We first give a brief overview of the equations and then look at simpler laws about the underlying joinad operations that are imposed by these equations. Many joinads are also monads, so the equations (3) and (4) relate `match!` to operations that are provided by monad (namely *map* and *join* used by `let!` and *unit* that enables `return`).

1. *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.

2. *Match first.* In ML-style pattern matching, we can have overlapping patterns and the compiler can identify unreachable clauses. This equation provides similar guarantees for the `match!` construct. The equation matches on a single computation, so it talks only about the *choose* operation.

3. *Correspondence to binding.* When the computation provides the `let!` construct, the meaning of `match!` in the degenerated case should be the same as the meaning of `let!` This equation describes a relation between *choose* and monadic *join.*

4. *Matching on units.* If the computation is a monad and provides the *unit* operation, we can specify the meaning of matching on two unit computations. This equation specifies an important aspect of *merge* operation.

As already mentioned, joinad needs to provide the *map* operation. This is common to all functors and follows usual laws [9], so we only discuss laws specific to joinads.

4.1 Merge operation laws

The laws that should hold about the *merge* operation are shown below. The first two laws follow from the equation 1 (reordering of arguments). The last law should hold only when the computation is a monad. In that case, the third law is required by the equation 4 (matching on units).

$$\begin{array}{rcll}
u \mathbin{⦷} (v \mathbin{⦷} w) & \equiv & \text{map assoc } ((u \mathbin{⦷} v) \mathbin{⦷} w) & \text{(associativity)} \\
u \mathbin{⦷} v & \equiv & \text{map swap } (v \mathbin{⦷} u) & \text{(commutativity)} \\
\text{unit } (a, b) & \equiv & (\text{unit } a) \mathbin{⦷} (\text{unit } b) & \text{(unit merge)}
\end{array}$$

$$\begin{array}{rl}
\textbf{where} & \text{assoc } ((a, b), c) = (a, (b, c)) \\
 & \text{swap } (a, b) = (b, a)
\end{array}$$

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 equation 1 will hold. The commutativity law reveals an interesting connection with commutative monads as discussed in section 5.

The third law (*unit merge*) specifies how the *merge* operation behaves with respect to monadic *unit*. In general, we cannot say anything about matching on multiple computations, so this law provides some cue in the case when the computation is a monad. We can apply the law to the equation 4 to get an equation that uses `match!` with only a single argument. The rest of the equation follows from the fact that *choose* is a generalization of the monadic *join* (as discussed in section 4.2). It may be of interest that this law is very similar to the *product law* of causal commutative arrows [24].

4.2 Choose operation laws

The equation 2 (match first) talks almost directly about the *choose* operation, but we can express it in simpler terms. The equation 3 (correspondence to binding) shows a property that must hold when a joinad computation also forms a monad. The laws about *choose* are less obvious due to the complexity of the operation:

$$\begin{array}{rll}
 & \text{choose [map } (\lambda v \to \text{Some } \mathit{expr}_1)\ m; & \\
 & \qquad\quad\ \text{map } (\lambda v \to \text{Some } \mathit{expr}_2)\ m\ ] & \text{(ordering)} \\
\equiv & \text{choose [map } (\lambda v \to \text{Some } \mathit{expr}_1)\ m\ ] &
\end{array}$$

$$\text{join} \equiv \text{choose [map } (\lambda v \to \text{Some } v)\ ] \qquad \text{(correspondence)}$$

The ordering law is essentially the result of direct translation of the equation 3. It specifies that the order of elements in the list given as the argument to *choose* matters. In particular, when there are multiple clauses that always succeed, the body of the first clause will be used. Notably, this law doesn't hold for proposals based on the `MonadPlus` typeclass [11, 13]. However, we believe that this property of ML-style pattern matching is essential for pattern matching on computations as well.

The correspondence law is applicable only when the computation in question is also a monad meaning that it defines operations *join* and *unit* in addition to *map*, *choose* and *merge*. This is a very important special case that deserves our attention. As mentioned in section 3.1, the *choose* operation bears similarity with monadic *join*. The type of the argument of *choose* is `list<M<Option<M<'T>>>>`, while the type of *join* is just `M<M<'T>>`. The correspondence law essentially says that the natural restriction of *choose* to a compatible type is equivalent to *join*.

5 Related notions of computations

In this section, we discuss the relationship between joinads and monads. We also discuss an interesting special case when a computation is joinad and a commutative monad. Due to the space restrictions, we do not cover relationships with idioms (also called applicative functors), which use an operation similar to our *merge*, but with a different set of laws. The thesis [4; Ch. 5] contains more information on this topic.

5.1 Relation with monads

When the computation is a monad, it needs to follow a set of monad laws that can be formulated in terms of *join*, *map* and *unit* (see for example [20]). As we saw earlier, if a joinad is also a monad, the *join* operation can be expressed in terms of *choose*. This means that a computation which is both joinad and monad can be defined just in terms of *choose*, *merge*, *map* and *unit*.

In that case, the computation needs to obey the laws of joinads (discussed in the previous section), but also the laws of monads [20]. We need to reformulate monad laws that involve *join* in terms of *choose*, but this can be easily done by replacing *join* with the definition from the correspondence law.

5.2 Commutative monads

Judging just from the type signature, it appears that the *merge* operation could be implemented in terms of *bind* and *unit* in a monad. We would use *bind* to obtain values of both of the arguments in a sequence and then use *unit* to return a tuple. This definition has the right type, but if we look at the merge laws, we find a problem.

The commutativity law of joinads states that reordering the arguments of *merge* should not change the meaning of code. This is not, in general, true for the implementation described above. However, if the monad is commutative, we can change the order of bindings and as a result, the described implementation is correct. A more detailed discussion including a proof can be found in [4].

In a retrospective on Haskell, Peyton-Jones considered working with commutative monads as an interesting open problem [15]. Although they are not sequential, the do-notation in Haskell [18] allows only a sequential use. Our work makes it possible to write code that works with commutative monads using `match!` in a less sequential fashion. If we have four values of type `Option<float>` representing possibly missing values that specify a location of a rectangle, we can calculate the center as follows:

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

We cannot write the calculation directly because the values are not numeric types. We first need to extract their content. Using `match!` we can obtain values of all four computations at once. In commutative monads, the order doesn't matter, so the arguments to `match!` can be rearranged in any way. The syntax still requires rebinding of all symbols, but it offers an interesting alternative to the do-notation.

6 Related work

We describe operations and laws of abstract computation type that makes it possible to pattern match on computations when composing computations. We discussed how our work relates to monads [2] and in particular commutative monads. Other related computation types include applicative functors [16] and arrows [22, 24]. We believe that it may be interesting to consider whether a generalized pattern matching could be provided for these computation types as well.

The existing work on pattern matching mainly focused on providing better abstraction when pattern matching on standard values [14, 21]. Extensible patterns in Scala [19] can be composed using custom operators. Some authors propose a generalization based on `MonadPlus` typeclass. This is an interesting alternative to our work, but it does not obey all equations that we intuitively expect (as discussed in section 4).

7 Conclusions

The key claim of this paper is that a range of important modern programming models can be encoded using a simple, retargetable and theoretically well founded extension. We describe an abstract computation *joinad* and present a lightweight syntax that makes it easy to write computations based on joinads. We use it for encoding declarative programming models for concurrent, reactive and parallel programming.

Our extension is based on pattern matching and we made a special effort to preserve the user's existing intuition about pattern matching. By requiring several laws about basic operations, we guarantee that usual reasoning about pattern matching applies in our generalized scenario. Finally, joinads are related to monads and in particular commutative monads which are considered as an interesting open problem.

We show that our construct can be used for binding on multiple monadic values in a less sequential fashion than the one provided by the usual do-notation.

Acknowledgements. We thank to Simon Peyton Jones, Gregory Neverov, Dmitry Lomov and James Margetson as well as anonymous reviewers of this and earlier version of this paper for useful comments and suggestions. Tomas is grateful to Microsoft Research for an internship invitation, which made this work possible.

References

1. Syme, D., Granicz, A., Cisternino, A.: Expert F#, Chapter 9. *Apress, 2007.*
2. Wadler, P.: Monads for functional programming. In LNCS Vol. 925, 1995.
3. Syme, D.: F# Language Specification. http://tinyurl.com/fsspec
4. Petricek, T.: Reactive Programming with Events (Master thesis), Charles University, 2010
5. Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In POPL 1996.
6. Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: JoCaml: A language for concurrent distributed and mobile programming. In LNCS Vol. 2638, pp 129–158. Springer, 2002.
7. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. ACM Trans. Program. Lang. Syst, 26(5):769–804, 2004.
8. Russo, C.: The Joins concurrency library. In PADL 2007.
9. Yorgey, B.: The Typeclassopedia. The Monad.Reader Issue 13. http://tinyurl.com/tycls
10. Haller, P., Van Cutsem, T.: Implementing Joins using Extensible Pattern Matching. In Proceedings of COORDINATION 2008.
11. Tullsen, M.: First class patterns. In Proceedings of PADL 2000
12. Fluet, M., Rainey, M., Reppy, J., Shaw, A.: Implicitly-threaded parallelism in Manticore. In Proceedings of ICFP 2008
13. Syme, D., Neverov, G., Margetson, J.: Extensible Pattern Matching via a Lightweight Language Extension. ICFP, 2007.
14. Wadler, P: Views: A way for pattern matching to cohabit with data abstraction. POPL 1987
15. Peyton Jones, S.: Wearing the hair shirt - A retrospective on Haskell. Invited talk, POPL 2003. Slides available online at: http://tinyurl.com/haskellretro
16. McBride, C. and Paterson, R.: Applicative programming with effects, Journal of Func. Programming 18 (2008)
17. Scholz, E.: Imperative streams - a monadic combinator library for synchronous programming. In Proc. ICFP, 1998
18. S. Peyton Jones (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, 2003.
19. Emir, B., Odersky, M., Williams, J.: Matching Objects with Patterns. ECOOP 2007
20. King, D., Wadler, P.: Combining Monads. In Proceedings of Glasgow Workshop on Functional Programming, 1992.
21. Okasaki, C.: Views for Standard ML. In Proc. of Workshop on ML, pp. 14–23, 1998.
22. Hughes, J.: Generalising Monads to Arrows, in Sci. of Comput. Prog. 37, pp. 67-111, 2000.
23. Elliott, C.: Declarative event-oriented programming. In Proceedings of PPDP 2000
24. Liu, H., Cheng, E., Hudak, P.: Causal commutative arrows and their optimization. ICFP '09
25. Bierman, G. M., Meijer, E., Torgersen, M.: Lost In Translation: Formalizing Proposed Extensions to C#. In Proc. of OOPSLA 2007
26. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus, Technical Report EDI-INF-RR-1258, School of Informatics, University of Edinburgh, 2008
27. Petricek, T., Syme, D.: Joinads (Extended version). Online at: http://tinyurl.com/joinads