Syntax Matters: Writing abstract computations in F#

Tomas Petricek[1], Don Syme[2]

[1] University of Cambridge, Cambridge, United Kingdom
[2] Microsoft Research, Cambridge, United Kingdom
tp322@cam.ac.uk, dsyme@microsoft.com

Abstract. The academic literature describes a number of abstract computation types such as monads, monoids, applicative functors and their compositions. These can be used to describe features of mainstream languages such as generators in Python or asynchronous computations in C# 5. Functional programmers are used to work with abstractions directly, but this is often difficult without a convenient syntactic sugar.

We give an overview of *computation expressions*, which is a syntactic sugar for working with abstract computations in F#. Unlike the `do` notation in Haskell, computation expressions are not tied to a single kind of abstract computations. They support wider range of computations, depending on what operations are available and they also provide greater syntactic flexibility.

As a result, F# programmers are able to use a single syntactic sugar for a wider range of computations including monoidal sequence generators, monadic parsers and applicative formlets. This removes the need for ad-hoc language extensions that provide "nice syntax" for one particular kind of computations.

1 Introduction

Abstract computation types like monads [1] provide a way for composing computations with some additional aspects, but monads are not the only example. Applicative functors [2] provide a weaker (thus more general) abstraction useful for web programming [18], while `MonadPlus` [8] is a stronger abstraction useful for parsers [10].

In Haskell, we can write such computations using a mix of combinators and syntactic extensions like monad comprehensions [19] and `do` notation. On the other hand, languages such as Python and C# emphasize the syntax and provide single-purpose support for asynchrony [20] or list generators [11]. Ideally, we would like to get the best of both worlds. A language should provide unified syntax that can capture different abstractions and enable appropriate syntax depending on the operations provided by the abstract computation type.

We argue that F# computation expressions provide such mechanism. Although the technical aspects of the feature have been described before [17], this paper is the first specific description of the breadth of applications of computation expressions from the perspective of abstract computations. The main contributions are:

Abstract computations. We describe abstract computations that can be written using F# 2.0 computation expressions. Aside from standard computations like monads and monoids (Sections 3 and 4), we show how to provide convenient notation for additive monads, monad transformers (Section 5) and applicative functors (Section 6).

Practical examples. Commonly used computation expressions match well-known computation types. Applications include asynchronous programming (Section 3.1), parsers, asynchronous sequences (Section 5) and applicative formlets (Section 6).

Handling of effects. F# is an impure language, so expressions may have effects. We reflect on how computation expressions embed (untracked) effects in abstract computations. We identify two approaches (Section 3.2) for different kinds of monads.

Examples in Sections 2 to 4 are based on F# 2.0 and Section 6 uses a research extension proposed by Petricek [22]. An upcoming version F# 3.0 extends the mechanism further to accommodate query syntax, but we leave that to future work. The examples in the paper focus on the breadth, so we omit implementations of the computations. These can be found in an online appendix at: http://tryjoinads.org/computations

2 Computation expressions

Computation expressions are blocks representing non-standard computations – that is, computations that have some additional aspect, such as laziness, asynchronous evaluation, hidden state or other. The code inside the block mirrors the standard F# syntax, but it is re-interpreted in the context of a non-standard computation. Computation expressions may also include a number of constructs that provide non-standard alternatives of standard constructs. For example, the `let!` syntax provides non-standard (monadic) version of `let` binding.

In this section, we use two examples to show how computation expressions unify single-purpose extensions from other languages. Then we look at the formal definition in the F# specification [17].

2.1 Unifying computations

The support for asynchronous programming in C# 5.0 is inspired by F# computation expressions, but it serves as a good example of single-purpose extension. The following C# 5.0 code downloads a specified URL without blocking the calling thread. It returns a `Task<string>` object that can be used to register callback that will be called when the operation completes:

```
async Task<string> GetLength(string url) {
  var html = await DownloadAsync(url);
  return html.Length;
}
```

The code uses standard C# features including `var` for variable binding and `return`. It also uses non-standard construct `await`, which specifies that `DownloadAsync` returns an asynchronous I/O operation and that the rest of the code should be run when the download completes. In F#, the body can be written as follows:

```
async { let! html = downloadAsync(url)
        return html.Length }
```

Computation expressions are enclosed in a block like `async { .. }` that determines the meaning of the computation. The `async` identifier is an object, which exposes members that are used to build the computation. Depending on the available members different keywords, written in bold font, are enabled (see Section 2.2).

The previous snippet uses non-standard `let` binding, written as `let!`, to denote the fact that the download is done asynchronously. This operation is interpreted using the `Bind` member of the `async` object. This example closely corresponds to the `do` notation for monads in Haskell [8], but we will see that we can use many other constructs.

As a second example, we use Python sequence generators that are used to construct lists. The following Python function duplicates elements in a list and multiplies the second occurrence of every element by 10:

```
def duplicate(list):
  for n in list:
    yield n
    yield n * 10
```

Haskell monad comprehensions [19] allow us to write `[ n * 10 | n <- list ]` to multiply all elements by 10, but they are not expressive enough to capture duplication. To do that, the code also needs monoidal operation to concatenate two lists. This can be done using the `MonadPlus` type class and `mplus` operation, but then we cannot benefit from any syntactic extension. In F#, we can write the sample as follows:

```
seq { for n in numbers do
        yield n
        yield n * 10 }
```

The `seq { .. }` block is again a computation expression, but it uses a different syntax, which is more appropriate for generators. As already noted, this example requires monoidal structure, which is provided by `Combine` member of `seq`. When the member is defined, the computation can produce multiple results using either `yield` or `return`.

2.2 Computation expressions

Computation expressions have been available in F# since 2007 [15] and they are fully documented in the F# language specification [17]. In this paper, we look at this purely syntactic language mechanism from a different perspective, but this section gives a brief overview of how the syntactic mechanism works. Showing the entire syntax and translation rules is beyond the scope of this paper, but we include most of the constructs that are used in this paper. Computation expression is an F# expression, but the body of computation expressions is a separate syntactic category:

expr	=	*expr* { *cexpr* }	Computation expression
cexpr	=	**let** *pat* = *expr* **in** *cexpr*	Binding value
	|	**let!** pat = expr **in** cexp	Binding computation
	|	**return** *expr*	Return value
	|	**return!** *expr*	Return computation
	|	**yield** *expr*	Yielding value
	|	**yield!** *expr*	Yielding computation
	|	**for** *pat* **in** *expr* **do** *cexpr*	For loop computation
	|	**while** *expr* **do** *cexpr*	While loop computation
	|	$cexpr_1$; $cexpr_2$	Composing computations
	|	*other-expr*	Effectful expression

The syntax includes `return`, `yield`, `let!` and `for` that were used in the previous section as well as a number of other constructs. The last two cases are perhaps more interesting. The first one represents a sequencing (or composition) of two computations. As we will see, the construct has different meanings for different types of abstract computations. The last one represents an effectful expression that can be turned into a computation that does not return a result, but performs some effect. Finally, the `in` keyword in `let!` and `let` can be omitted when using a line break instead.

A concrete computation block such as `async { .. }` or `seq { .. }` does not allow all of the keywords – a construct can only be used if the static type of the object (`async` or `seq`) defines members that are required by the translation of the construct. The object `async` or `seq` is called *computation builder*. Assuming `bind` and `unit` are operations of the asynchronous workflow monad, a computation builder `async` that enables `let!` and `return` syntax can be written as follows:

```
type AsyncBuilder() =
  member x.Bind(ma, f) = bind f ma
  member x.Return(a) = unit a

let async = AsyncBuilder()
```

The type `AsyncBuilder` is an F# class with members `Bind` and `Return`. To write a computation expression block `async { .. }`, we create an instance of the object and assign it to the `async` identifier. In the rest of the paper, we do not use the actual computation builder notation and show just member names together with their types[1].

As mentioned, computation expressions are a lightweight syntactic sugar. They are translated before type-checking, according to the rules shown in Figure 1. The fact that the translation occurs before type-checking is significant. It allows more flexibility in definitions of the operations – for example, the `Combine` member has different type for monads (Sections 3.2) and monoids (Section 4).

The translation is defined as a function $[\![-]\!]_{\mathrm{m}}$ parameterized by the computation builder name. The identifier is used to generate member calls such as `m.Bind` (translation of `let!`) or `m.Return` (translation `return`). The omitted rules for `yield!` and `yield` are similar to `return` and `return!` using member names `Yield` and `YieldFrom`.

[1] Polymorphic types are written using a lightweight notation *ma* instead of the standard F# notation `M<'T>`. For example, the type of monadic bind is written as $ma \to (a \to mb) \to mb$.

$$
\begin{aligned}
\mathit{expr}\ \{\ \mathit{cexpr}\ \} &= \textbf{let}\ \mathrm{m} = \mathit{expr}\ \textbf{in}\ \Delta[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} \\
[\![\ \textbf{let}\ \mathit{pat} = \mathit{expr}\ \textbf{in}\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= \textbf{let}\ \mathit{pat} = \mathit{expr}\ \textbf{in}\ [\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} \\
[\![\ \textbf{let!}\ \mathit{pat} = \mathit{expr}\ \textbf{in}\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= \mathrm{m.Bind}(\mathit{expr}, \textbf{fun}\ \mathit{pat} \rightarrow [\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}}) \\
[\![\ \textbf{return}\ \mathit{expr}\ ]\!]_{\mathrm{m}} &= \mathrm{m.Return}(\mathit{expr}) \\
[\![\ \textbf{return!}\ \mathit{expr}\ ]\!]_{\mathrm{m}} &= \mathrm{m.ReturnFrom}(\mathit{expr}) \\
[\![\ \textbf{for}\ \mathit{pat}\ \textbf{in}\ \mathit{expr}\ \textbf{do}\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= \mathrm{m.For}(\mathit{expr}, \textbf{fun}\ \mathit{pat} \rightarrow [\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}}) \\
[\![\ \textbf{while}\ \mathit{expr}\ \textbf{do}\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= \mathrm{m.While}(\mathit{expr}, \Lambda[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}}) \\
[\![\ \mathit{cexpr}_1\ \textbf{;}\ \mathit{cexpr}_2\ ]\!]_{\mathrm{m}} &= \mathrm{m.Combine}([\![\ \mathit{cexpr}_1\ ]\!]_{\mathrm{m}}, \Lambda[\![\ \mathit{cexpr}_2\ ]\!]_{\mathrm{m}}) \\
[\![\ \mathit{other\text{-}expr}\ ]\!]_{\mathrm{m}} &= \mathit{other\text{-}expr}; \mathrm{m.Zero}() \\
\\
\Lambda[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= \mathrm{m.Delay}(\textbf{fun}\ () \rightarrow [\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}}) \\
\Lambda[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= [\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} \qquad \text{(if Delay is missing)} \\
\Delta[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= \mathrm{m.Run}(\Lambda[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}}) \\
\Delta[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} &= [\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}} \qquad \text{(if Run is missing)}
\end{aligned}
$$

Fig. 1. Translation rules for computation expressions

A particular construct of computation expression syntax is allowed only when the static type of the computation builder defines members that are required by the translation. For example, it is not possible to use `let!` or `return` inside `seq { .. }` block, because `seq.Bind` and `seq.Return` are not defined.

An exception from this rule are `Delay` and `Run` members. These are used if they are provided, but when they are not available, a different translation is used. The use of these two operations is captured by helper functions $\Delta[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}}$ and $\Lambda[\![\ \mathit{cexpr}\ ]\!]_{\mathrm{m}}$. The first one applies `m.Run` to the translated computation expression (in the first rule) and the second wraps translated computation in a lambda function and calls `m.Delay` (in arguments of `Run`, `While` and `Combine`).

Although F# computation expressions are purely syntactic, they are used to program with well known abstract computations. Often, there are multiple encodings using different syntax. Analyzing and describing these is our key contribution.

3 Monads

Monads, introduced by Moggi [5] and popularized by Wadler [1], are the most widely known class of abstract computations. In purely functional languages, they are useful for propagating state, exceptions or effects [6]. The most prominent monads in F# are *asynchronous workflows* [7]. They are used for writing non-blocking code that involves long-running I/O operations without explicit use of callbacks.

3.1 Functional monads

Asynchronous workflows use the most common syntax for writing monadic computations. Monadic binding is written using `let!` and returning of a value using `return`. To allow the syntax, computation builder must provide the following operations:

return : $a \to ma$
bind : $ma \to (a \to mb) \to mb$

Defining *bind* also enables the `do!` keyword, which is used to call asynchronous operations that return `unit`. The following code downloads a web page (`asyncDownloadUrl` returns asynchronous computation), waits 1 second and returns the length of the page:

```
async { let! html = asyncDownloadUrl url
        do! Async.Sleep(1000)
        return html.Length }
```

The `do!` notation is treated as a monadic binding that matches the result of computation against the unit pattern and it is equivalent to writing `let! () = e`. The above snippet is translated using two nested calls to *bind* and a single call to *return*:

```
async.Bind(asyncDownloadUrl url, fun html ->
  async.Bind(Async.Sleep(1000), fun () ->
    async.Return(html.Length)))
```

The `return` construct is used to lift a value into a monad, but we also need syntax for returning the result of an existing monadic computation. To keep the syntax uniform, F# uses the `return!` keyword (`let` binds a value, `let!` binds a computation, `return` returns a value and `return!` returns a computation).

To download a web page asynchronously and immediately return the result, we can write `return! asyncDownloadUrl url`. The translation requires the following member:

returnFrom : $ma \to ma$

The only purpose of *returnFrom* is to allow the `return!` keyword, which is not desirable for some computations (i.e. in Section 4). Although the operation can implement some behavior, it is usually defined to be just an identity function.

The syntax introduced so far is very similar to the `do` notation in Haskell [8]. The next section shows additional constructs that are allowed in monadic syntax in F#.

Monad laws. Monad laws specify that certain syntactic transformations preserve the meaning of code. These are defined in terms of *bind* and *return*, but they can also be expressed using the syntax – monad comprehensions or `do` notation [1]. We do not show the syntactic equivalences using the F# computation expression syntax, because they are quite similar to the well-known equivalences using the Haskell `do` notation.

3.2 Monads with sequencing and effects

F# uses the `unit` type for primitive effectful computations. The sequencing expression `e1; e2` first evaluates `e1` which is required to return `unit`, then evaluates `e2` and returns its result. In addition, F# also allows an `if` expression without the `else` branch (`if e1 then e2`), which requires `e2` to return `unit` and implicitly returns `unit` in the `false` case.

Monadic sequencing and unit can be expressed in terms of *bind* and *return*, but that excludes other uses of the syntax (Section 5), so the translation requires additional operations. For monads, these can be defined in two ways.

Sequencing monadic computations. If *ma* represents a computation that can be executed later, it is possible to define a *delay* operation that takes an effectful function and wraps it in an *ma* value. Aside from *delay*, we define the following operations:

delay	$: (1 \to ma) \to ma$	combine	$: m1 \to ma \to ma$
zero	$: 1 \to m1$	run	$: ma \to ma$

For monadic computations, it is possible to define the above operations in terms of *bind* and *return*, but we leave that as an exercise. The *zero* operation represents a monadic unit value and *combine* corresponds to the `;` operator. The *run* operation wraps the entire computation expression (which is returned by *delay*) and can be defined as an identity function. We will see its utility in the next section.

The operations are used when translating a computation expression that does not return a value (loops or `if` without the `else` branch), followed by another construct:

```
async { if delay then do! Async.Sleep(1000)
        return! asyncFetch url }
```

If `delay` is `true`, the workflow waits for one second. Then it asynchronously downloads a web page and returns the result. The code is translated as follows:

```
async.Run(async.Delay(fun () ->
  async.Combine(
    ( if delay then
        async.Bind(Async.Sleep(1000), fun () -> async.Zero())
      else async.Zero() ),
    async.Delay(fun () -> async.ReturnFrom(asyncFetch url)) )))
```

The `true` branch is translated as a binding that returns the *zero* computation. The false branch is added, returning the *zero* computation as well. The result is then combined with the upcoming *returnFrom*, which is wrapped in *delay* to avoid performing effects too early when evaluating the arguments of *combine*. The first argument of *combine* is a unit-returning computation. As a result, it is not possible to use `return` imperatively to jump out of a monadic computation.

The translation wraps the whole computation in a lambda function passed to *delay* and then applies *run* on the result. As with sequencing, the overall *delay* wraps any immediate effects (in the conditional) inside the computation.

Sequencing monadic containers. If *ma* represents a wrapped, non-delayed value (such as `option<'T>`) we cannot implement *delay* of the previous type without performing the effects. However, the translation allows different typing of the operations:

delay	$: (1 \to ma) \to (1 \to ma)$	combine	$: m1 \to (1 \to ma) \to ma$
zero	$: 1 \to m1$	run	$: (1 \to ma) \to ma$

In this alternative, *delay* may be an identity function that simply returns the provided function. The result of *delay* is passed as a second argument to *combine* (and to *run*), so we modify their types accordingly. The operations *zero* and *combine* can be implemented by *return* and *bind*. Finally, *run* applies the delayed function (potentially performing the effects) to get the result of a computation. This variant makes it possible to sequence computations of monads representing containers, such as Maybe:

```
maybe { if b = 0 then return! fail()
        return a / b }
```

The translation for this snippet calls *combine* with translation of the first and second line as the first and second argument, respectively. If *delay* evaluated the function to obtain an *ma* value, it would evaluate `a / b` even if `b` equals zero. By returning a function, the *combine* operation may not need to evaluate the second computation (if the first one fails), avoiding a runtime exception caused by division by zero.

Unifying delayed computations. So far, we represented delayed computations using $\boldsymbol{ma}$ (for *computations*) and $\mathbf{1} \to \boldsymbol{ma}$ (for *containers*). We can generalize the two cases by using a new abstract type $\boldsymbol{da}$ for delayed computations:

$$\begin{array}{ll} \text{delay} : (1 \to ma) \to da & \text{combine} : m1 \to da \to ma \\ \text{zero} : 1 \to m1 & \text{run} : da \to ma \end{array}$$

To our knowledge, the only types used for da in practice so far are ma and $1 \to ma$. However, the generalization simplifies the discussion in upcoming sections. The computation da may itself have a monadic structure, but this is not required. We only require that delaying an effect-free function and evaluating it using *run* is an identity. Formally $run\ (delay\ (\lambda x.n)) = n$, where n is some effect-free computation.

The equation holds for the standard definitions of *delay* and *run*. It guarantees that the added operations do not change the meaning in case when they are not needed.

3.3 Monadic control flow constructs

The syntax allowed inside computation expressions aims to provide computation-specific versions of most of the standard control flow constructs of F#. In this section, we look at the support for (imperative) loops and exception handling.

Looping syntax. The computation expression syntax supports looping constructs `for` and `while`. For monads, these can be defined using *zero*, *combine* and *bind*, but the translation allows other useful definitions (Sections 4 and 5.2):

$$\begin{array}{ll} \text{for} & : [a] \to (a \to m1) \to m1 \\ \text{while} & : (1 \to \text{bool}) \to d1 \to m1 \end{array}$$

The *for* operation represents sequencing of computations generated from a list and *while* represents repeated evaluation of a computation while a condition (relying on mutable state) holds. The first argument is a function that evaluates the condition. The second argument represents a delayed body. In *for*, the second argument is always a function, so *delay* is not used.

In asynchronous workflows, looping constructs are useful for writing repeating and long-running computations. The following example is adapted from [7]:

```
async { while true do
          for color in [green; orange; red] do
            do! Async.Sleep(1000)
            displayLight color }
```

The code creates a workflow that repeatedly changes the color of a semaphore light with a 1 second delay. The function `displayLight` mutates the user interface.

Monadic exception handling. In an impure language that supports exceptions, it is important to provide a mechanism for exception handling within the monadic syntax. Doing that manually would require wrapping every sub-expression using an exception handler, because the translation introduces new scopes.

The handling of exceptions in F# is delegated to *tryWith* and *tryFinally* members that represents a monadic versions of `try .. with` and `try .. finally` expressions:

tryWith : $da \to (\text{exn} \to ma) \to ma$
tryFinally : $da \to (1 \to 1) \to ma$

The first argument is a computation (obtained using *delay*) that represents un-evaluated body. The second argument of *tryWith* is an exception handler that takes a value representing the exception (*exn*) as an argument. The second argument of *tryFinally* is a cleanup function that releases resources allocated in the current scope.

In the case when $da = 1 \to ma$ and ma represents a fully evaluated computation, the two operations only need to handle exceptions triggered by evaluation of the delayed computation, so their implementation is straightforward. In the case when $da = ma$ and ma represents a computation itself, the monadic type needs to provide a mechanism for exception handling. For example, asynchronous workflows use an exception continuation for reporting exception, which is used by *tryWith* and *tryFinally*.

4 Semigroups and monoids

Other structures than monads that are ubiquitous in functional programming are semigroups and monoids. A semigroup consists of a set of values S and an associative operator $\circ$. A monoid is a semigroup that also includes a special unit element e.

Well known examples of semigroups or monoids are natural numbers (with multiplication and 1 or addition and 0), Booleans (with conjunction and `true`), lists (with concatenation and empty list), but also Maybe (with left-biased operation for combination). Most of these structures are monoids, so we do not discuss semigroups separately, but we mention what operations would not be available for a semigroup.

Monoids and semigroups. To use computation expression syntax for a monoid, we need to define the operations below. Similarly to the handling of effects in monads, encoding of monoidal computations requires *delay* of type $(\mathbf{1} \to \boldsymbol{ma}) \to \boldsymbol{da}$, where $\boldsymbol{da}$ is typically either $\boldsymbol{ma}$ for computations or $\mathbf{1} \to \boldsymbol{ma}$ for containers.

combine	$: ma \to da \to ma$	run	$: da \to ma$
zero	$: 1 \to ma$	delay	$: (1 \to ma) \to da$
yield	$: a \to ma$	yieldFrom	$: ma \to ma$

The *delay* and *run* operations have the same type and the same purpose as previously for monads. The key operations that define the monoidal structure are *combine*, *zero*

and *yield*. The *combine* function represents the binary operation of the monoid. As F# is an eager language, the second argument of *combine* needs to be delayed, so that effects that might happen when evaluating it only happen at the time when the computation is required (i.e. when evaluating n^{th} element of a lazy list).

The *yield* operation is similar to *return*. It is used to build primitive computations of the monoid (elements of the set S). Defining *yield* instead of *return* means that the computation uses `yield` syntax instead of `return`. Although this is just a syntactical difference, the name hints that the computation can produce multiple results. We also include *yieldFrom*, which allows composing computations using `yield!`

Finally, *zero* is the unit of the monoid. Its type is ma, in contrast with $m1$ that was used for monads. This shows that it plays a different role – instead of representing unit computation, it represents a computation that does not contain a value.

Delayed monoidal computations. As an example, consider a monoid formed by integers with multiplication and unit 1. Using a computation builder with the above operations, we can calculate factorial of 10 using `factorial 0`:

```
let rec factorial n =
  mul { yield n
        if n <= 10 then yield! factorial (n + 1) }
```

The body uses *combine* to compose computation that returns a singleton list (*yield*) with a computation that generates the remaining elements. The second argument is wrapped using *delay*, so that the entire list is not evaluated immediately:

```
mul.Run(mul.Delay(
  mul.Combine(mul.Yield(n), mul.Delay(fun () ->
    if n <= 10 then mul.YieldFrom(factorial (n + 1))
    else mul.Zero())) ))
```

Integers are not delayed computations, so *delay* returns a function that is later executed by *run*. However, the same translation would work for lazy computations.

Control flow constructs for monoids. Similarly to monads a computation with a monoidal structure can provide additional members to enable standard F# control flow constructs. Defining the following in terms of *zero* and *combine* is left as an exercise:

$$\begin{aligned} &\text{for} && : [a] \to (a \to mb) \to mb \\ &\text{while} && : (1 \to \text{bool}) \to da \to ma \end{aligned}$$

The operations have different types than when working with monads (Section 3.3). The body of the *for* loop can now return values (type $1 \to mb$ instead of $1 \to m1$) and so can the body of *while* (da instead of $d1$). Using the `mul` computation builder with these additional members, we can calculate a factorial as follows:

```
mul { for num in 1 .. 10 do yield num }
```

Monoid laws. Every monoid is required to obey two laws. The binary operation must be associative ($a \circ (b \circ c) = (a \circ b) \circ c$) and the unit element is required to behave as

a unit ($1 \circ a = a = a \circ 1$). Similarly to the monad laws, the properties of monoids map to syntactic equalities about monoidal computation expressions:

m { *cexpr*$_1$; *cexpr*$_2$; *cexpr*$_3$ } ≡ m { **yield!** m { *cexpr*$_1$; *cexpr*$_2$ }; *cexpr*$_3$ }

m { **if false then** *cexpr*$_1$ *cexpr*$_2$ } ≡ m { *cexpr*$_2$ } ≡ m { **if false then** *cexpr*$_1$ *cexpr*$_2$ }

Both of the equalities are intuitively expected to hold when working with monoidal computations. The first corresponds to the associativity and the second to the unit laws. The *zero* of monoid does not have a direct correspondence in the syntax, but one way to obtain it is to use the `if` expression without the `else` clause.

5 Composed computations

We showed that computation expressions provide a notation for monads and monoids. In this section, we take one step further – we look at computations that combine monoids and monads and also computations formed by a layer of monads.

5.1 Additive monads

An additive monad is a monad with a monoid structure, also known as `MonadPlus` in Haskell. Common examples are parser combinators and collections. Both of these can be encoded in F#, but the desirable syntax differs.

Parser combinators. Monadic parsers [10] provide both monadic and monoidal interface. The *unit* operation represents a parser that always succeeds without consuming any input and *bind* represents sequencing (where the second parser may depend on the value parsed first). The monoid structure defines *combine* as either left-biased or non-deterministic choice and *zero* represents a failing parser.

When defining additive monads, we need to choose whether to use the `return` or the `yield` keyword. This is a matter of style, but it depends on which interface is considered more important. For parsers, we prefer monad and define *return*:

return	$: a \to ma$	zero	$: 1 \to ma$
bind	$: ma \to (a \to mb) \to mb$	combine	$: ma \to da \to ma$

The type of *bind*, *return* and *zero* are the same as for monads and monoids previously. We omit *run*, *delay* and *returnFrom* which are standard. The type of *combine* follows the definition used by monoids (with arguments ma and da), which is more general than the one for monads (with $m1$ as the first argument). Sample parsers that recognize one or more and zero or more repetitions a predicate `p` are written as follows:

```
let rec oneOrMore p = parse {
  let! x = p
  let! xs = zeroOrMore p
  return x::xs }
```

```
and zeroOrMore p = parse {
  return! oneOrMore p
  return [] }
```

The definition of `oneOrMore` uses monadic features to say that the parser should parse `p` followed by zero or more repetitions of `p`. The definition of `zeroOrMore` is translated using a monoidal structure that provides a choice from two alternatives. The first is to parse one or more repetitions of `p` and the other is to succeed and return immediately.

Sequence expressions. Sequences (or lists) are another example of additive monads. The usual definition in F# uses syntax that is quite different from the previous section:

```
let rec listFiles dir = seq {
  yield! Directory.GetFiles(dir)
  for subdir in Directory.GetDirectories(dir) do
    yield! listFiles subdir }
```

The body combines all files generated from the current directory with a sequence that is generated by concatenating all files from sub-directories (recursively). The monoidal interface provides *combine* to concatenate collections (one generated by `yield!` and other by `for`). The monadic structure provides return to build a singleton list and bind, which concatenates all generated collections.

The above syntax emphasizes the monoidal structure and so it the *return* of a monad is exposed as *yield* and *bind* is implemented as *for*. In fact, the usual type of *for* for lists ($[a] \to (a \to mb) \to mb$) overlaps with the type of *bind*, because $ma = [a]$.

5.2 Layered monads

It is often desirable to combine non-standard aspects of multiple computation types. One example is to combine non-blocking execution of *asynchronous workflows* with the ability to return multiple results of *sequences*. For example, an asynchronous file download might return data in 1kB buffers. Such a computation type is implemented by asynchronous sequences [14].

Assuming `urls` is an asynchronous sequence of URLs (produced by the user input), the following snippet asynchronously produces the URLs together with their content:

```
let pages = asyncSeq {
  let wc = new WebClient()
  for url in urls do
    let! html = wc.AsyncDownloadString(Uri(url))
    yield url, html }
```

The result is a computation that can be called to obtain the head (information about the next page) together with the tail (that can be called again). On the first call, the computation initializes `WebClient`, obtains the first URL from the *asynchronous sequence* `urls`, downloads its content and returns it. The download is performed by binding on an *asynchronous workflow* returned by `AsyncDownloadString`. When the caller requests the next value, the computation runs the next iteration of the `for` loop.

Binding for layered monads. The layered monad used above combines an asynchronous workflow `Async<'T>` (written $A\alpha$) and an additive monad $[\alpha]$ into a composed type $AS\alpha$. Both of the basic types can be lifted to the composite type. We can:

- turn an async workflow into an async sequence that returns a single value, and
- turn a list into an asynchronous sequence that produces values immediately.

The bind operation of the composed monad takes $AS\alpha$ as an argument, but it is also possible to define operations that lift a list and asynchronous workflow, respectively, into an asynchronous sequence and then perform the binding. To provide a convenient syntax, we use ad-hoc polymorphism (overloading) for members and define:

$$\begin{aligned}&\text{for} &&: \mathrm{AS}a \to (a \to \mathrm{AS}b) \to \mathrm{AS}b\\ &\text{for} &&: [\mathrm{a}] \to (a \to \mathrm{AS}b) \to \mathrm{AS}b\\ &\text{bind} &&: \mathrm{A}a \to (a \to \mathrm{AS}b) \to \mathrm{AS}b\end{aligned}$$

This definition defines the `for` keyword (first *for*) as a monadic bind on asynchronous sequences. The second overloaded *for* allows using `for` keyword to iterate over lists and *bind* enables `let!` for calling asynchronous workflows (returning a single result).

The choice of the `for` keyword for binding on asynchronous sequences is made to match the expectations of the user – the body of the `for` loop is executed repeatedly (for both lists and asynchronous sequences), while the body of `let!` is executed just once (when binding on an asynchronous workflow).

Monad transformer laws. In Haskell, monads can be layered using monad transformers [13]. Among other applications, they can be used to add state or exceptions to other monads. F# does not support higher-kinded types, so monad transformers cannot be encoded directly, but they provide a useful framework for documenting computations. The above example can be viewed as an application of the list monad transformer [16] to the asynchronous workflow monad.

Monad transformers also specify a set of laws that should hold about composed computations. These are written in terms of *return* and *bind* of the underlying and resulting monads (`async` and `asyncSeq` in our example) and the *lift* operation that turns the underlying monad to the composed (type $A\alpha \to AS\alpha$). The laws can be expressed in the computation expression syntax as follows:

asyncSeq { **let!** x = async { **return** v } **in return** x } ≡ asyncSeq { **return** v }

asyncSeq { **let!** y = async { **let!** $x = m$ **in** $cexpr_1$ } **in** $cexpr_2$ }
≡ asyncSeq { **let!** $x = m$ in **let!** y = async { $cexpr_1$ } **in** $cexpr_2$ }

The `let!` syntax inside `async` block corresponds to the *bind* of the underlying monad. Inside `asyncSeq`, it corresponds to the *lift* operation followed by *bind* of the composed monad. There is no syntax corresponding only to *lift*. However, we can use the right unit law of monads (in the first equation), which guarantees that *bind* followed by *return* preserves the meaning of a computation.

6 Applicative functors

Applicative functors [2] provide a weaker abstraction than monads – every monad is an applicative functor, but not conversely. For example, formlets [18] are an applicative functor for composing HTML forms that are not monads. Using a weaker abstraction also allows more efficient implementation of parsers [25].

There are two ways of defining applicative functors. The first relies on an operation of type $m(a \to b) \to ma \to mb$ and is more suitable for writing computations in an applicative style and propagating their effects. We use the second style, which is more suitable for computational interpretation used, for example, by formlets:

$$\begin{aligned} &\text{merge} \quad : ma \to mb \to m(a, b) \\ &\text{map} \quad : ma \to (a \to b) \to mb \\ &\text{return} \quad : a \to ma \end{aligned}$$

The declaration replaces monadic *bind* with *map* and *merge*; *map* applies a function to values carried by the abstract computation and *merge* composes additional aspects (or effects) of two computations and combines their values using a tuple.

The difference between applicative functors and monads is that monadic computations can perform different effects depending on values obtained as the result. On the other hand, the additional (effect) structure of applicative functors is fully determined regardless of the calculations performed using *map*.

The next sample uses formlets to create a registration form consisting of a textbox and a dropdown. When the values are entered, it produces a string with user details:

```
let userFormlet = formlet {
  let! name = Formlet.textBox
  and gender = Formlet.dropDown ["Male"; "Female"]
  return name + " " + gender }
```

The computation describes two aspect of HTML form – the rendering and the processing of entered values. The rendering phase uses the fixed structure of the computation to produce HTML code representing the form. In the processing phase, the values of `name` and `gender` are available and are used to calculate the result of the form.

The structure of applicative computations cannot depends on the values, so the syntax for uses parallel binding (`let! .. and ..`), which binds a fixed number of independent computations. The rest of the computation cannot contain other bindings. The computation expression from the previous example is translated as follows:

```
formlet.Map
  ( formlet.Merge Formlet.textBox (Formlet.dropDown [ ... ]),
    fun (name, gender) -> name + " " + gender )
```

The parallel binding is turned into an expression that combines all bindings using the *merge* operation. This part of the computation defines the structure and formlets use it for rendering. The rest of the computation is translated into projection (by removing the `return` keyword) and is applied to the composed computation (formlet) using *map*. In formlets, the projection function is evaluated only in the input processing phase.

7 Related work

Abstract computations. Syntactic sugar exist for a number of abstract computations. Haskell monad comprehensions [19] and `do` notation are used for working with monads; McBride [2] proposes a notation for applicative functors. These are all quite different – in this paper, we have described a single syntax that captures a wider range of computations. However, our syntax is not flexible enough to encode arrows [21].

Delimited continuations. Filinski demonstrated [26] that monadic computations can be encoded using continuations. An intriguing question is whether this work could lead to a simpler notation for monads and more complex structures discussed in this paper. This alternative could be attractive for languages that support delimited continuations like Scala [9]. The `reset` operation of delimited continuations seems related to the *run* function of computation expressions, which can be also used to restrict the scope of behavior. This aspect is used by the imperative computation expression [12].

Generators and monads. Generators, also called iterators [23] are the most common class of non-standard computations in main-stream languages. Although they are designed specifically for collections, they have been used to encode other monadic computations [21] and also delimited continuations [24]. However, the fact that the syntax has been designed for another purpose is a limiting factor.

Languages with effects. As far as we are aware, the handling of effects allowed by a host language in monadic computations has not been discussed previously. However, our *delay* operation is similar to Filinski's *reify* operation [3]. Instead of capturing all effects, it combines effects of the function arrow and effects associated with the resulting computation. Using a precise type system with annotations for effects, such as [4], the type of *delay* would be written as:

$$\left(1 \xrightarrow{r} m^{\mathrm{s}} a\right) \xrightarrow{\mathrm{t}} m^{\mathrm{u}} a$$

The operation represents redistribution of effects. Assuming that $\oplus$ represents a combination of effects, it must hold that $r \oplus s = t \oplus u$. A default implementation of *delay* simply applies the function (giving $r = t \wedge s = u$), but for computations that can capture effects, it is desirable to provide operation where $(u = r \oplus s) \wedge (t = 0)$.

8 Conclusions

We presented F# computation expressions, which provide a unified syntactic sugar for working with a wide range of abstract computations. We showed that a single syntactic mechanism can be used for working with monoids, monads and applicative functors, as well as computations composed using monad transformers. We believe that an easy to use syntax is the key for making the expressivity and compositionality of these abstractions available to a wider range of practitioners.

References

1. Wadler, P.: Monads for functional programming. In LNCS Vol. 925, 1995.
2. McBride, C. and Paterson, R.: Applicative programming with effects, Journal of Func. Programming 18 (2008)
3. Filinski, A.: Monads in Action. In Proceedings of POPL 2012.
4. Wadler, P. and Thiemann, P.: The Marriage of Effects and Monads. In ACM Trans. Comput. Logic, vol. 4, num. 1, pp. 1-32, January 2003.
5. Moggi, E.: Notions of Computation and Monads. In Inf. Comput., vol 93, pp. 55-92, 1991
6. Peyton Jones, S. and Wadler, P.: Imperative Functional Programming. POPL, 1993.
7. Syme, D., Petricek, T. and Lomov, D.: The F# Asynchronous Programming Model. In Proceedings of PADL 2011.
8. Peyton Jones, S., et al.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, 2003, ISBN: 9780521826143
9. Rompf, T., Maier, I. and Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In Proceedings of ICFP, 2009
10. Hutton, G. and Meijer, E.: Monadic Parsing in Haskell. In J. Funct. Program., vol. 8, num. 4, pp. 437-444, July 1998
11. Hejlsberg, A., Wiltamuth, S., Golde, P.: C# Language Specification, Addison-Wesley, 2003
12. Petricek, T.: Imperative Computations in F# (I. and II.), Unpublished. Retrieved 23 March 2012. Available online at: http://tomasp.net/blog/imperative-ii-break.aspx
13. Liang, S., Hudak, P. and Jones, M.: Monad Transformers and Modular Interpreters. In Proceedings of POPL 1995.
14. Petricek, T.: Programming with F# Asynchronous Sequences, Unpublished. Retrieved 23 March 2012, Available online at: http://tomasp.net/blog/async-sequences.aspx
15. Syme, D.: Some Details on F# Computation Expressions. http://tinyurl.com/comp-expr
16. Haskell Wiki. ListT done right (Unpublished). Retrieved 23 March 2012, Available online at: http://www.haskell.org/haskellwiki/ListT_done_right
17. Syme, D. et al.: The F# 2.0 Language Specification (April 2010), Retrieved 23 March 2012, Available online at: http://tinyurl.com/fsharp-spec
18. Cooper, E., Lindley, S., Wadler, P. and Yallop, J.: The Essence of Form Abstraction. In Proceedings of APLAS, 2008.
19. Giorgidze, G., Grust, T., Schweinsberg, N. and Weijers, J.: Bringing Back Monad Comprehensions. In Proceedings of Haskell Symposium, 2011, Tokyo, Japan
20. Bierman, G., Russo, C., Mainland, G., Meijer, E. and Torgersen, M.: Pause 'n' play: Formalizing asynchronous C#. In Proceedings of ECOOP, 2012
21. Paterson, R.: A new notation for arrows. In Proceedings of ICFP, 2001
22. Petricek, T.: Variations in F#. Retrieved 24 March 2012, Available online at: http://tomasp.net/blog/fsharp-variations-joinads.aspx
23. Jacobs, B., Meijer, E., Piessens, F. and Schulte, W.: Iterators Revisited. In Proceedings of the 7th ECOOP Workshop on Formal Techniques for Java-like Programs, 2005
24. James, R. P. and Sabry, A.: Yield: Mainstream Delimited Continuations. In Proceedings of Theory and Practice of Delimited Continuations, 2011.
25. Swierstra, S. D.: Combinator Parsing: A Short Tutorial, In Language Engineering and Rigorous Software Development, pp. 252-300, 2009
26. Filinski, A.: Representing layered monads. In Proceedings of POPL, 1999