The F# Computation Expressions Zoo

Tomas Petricek[1] and Don Syme[2]

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

Abstract. Many computations can be structured using abstract types such as monoids, monad transformers or applicative functors. Functional programmers use those abstractions directly, but main-stream languages often integrate concrete instances as language features – e.g. generators in Python or asynchronous computations in C# 5.0. The question is, is there a sweet spot between convenient but inflexible language feature and flexible, but more difficult to use library?

F# *computation expressions* answer this question in affirmative. Unlike the do notation in Haskell, computation expressions are not tied to a single kind of abstraction. They support a wide range of computations, depending on what operations are available. They also provide greater syntactic flexibility leading to a more intuitive syntax.

We show that computation expressions can structure well-known computations such as monoidal list comprehensions, monadic parsers, applicative formlets and asynchronous sequences based on the list monad transformer. We also present typing for computation expressions that is capable of capturing all these applications.

1 Introduction

Structures like monads [1] provide a way for composing computations with additional features. There are many examples – monads can be composed using monad transformers [2], applicative functors provide a more general abstraction useful for web programming [3] and additive monads are useful for parsers [4].

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] and list generators [11].

We believe that syntax matters – a language should provide *uniform* syntactic support that can capture different abstractions, but is *adaptable* and enables appropriate syntax depending on the abstraction. This paper shows that F# computation expressions provide such mechanism.

Although the technical aspects of the feature have been described before[3] [17], this paper is novel in that it relates the mechanism to well-known abstract computations. We also present new typing based on those uses.

[3] F# 3.0 extends the mechanism further to accomodate extensible query syntax. To keep this paper focused, we leave analysis of these extensions to future work.

Practical examples. We demonstrate the breath of computations that can be structured using F# computation expressions. The applications include asynchronous workflows and sequences §2.1, §2.3, list comprehensions and monadic parsers §2.2 and formlets for web programming §2.4.

Abstract computations. We show that the above examples fit well-known types of abstract computations, including additive monads and monad transformers, and we show what syntactic equalities hold as a result §5.

Syntax and typing. We revisit the definitions of computation expressions. We provide typing rules that capture idiomatic uses §3.2, extend the translation to support applicative functors §6 and discuss the threatment of effects §4 that is needed in impure language.

We believe that software artifacts in programming language research matter [99], so all examples with implementations can be found and interactively run online: `http://tryjoinads.org/computations`. The syntax for applicative functors is a reserch extension; all other examples can be compiled with F# 2.0.

2 Computation expressions by example

Computation expressions are blocks of code that represent computation with some non-standard aspect such as laziness, asynchronous evaluation, hidden state or other. The code inside the block is re-interpreted using *computation builder*, which is a record of operations that define the computation. It also defines what syntax is available in the block[4].

Computation expressions mirror the standard F# syntax (let binding, loops, exception handling), but support additonal computational constructs. For example let! represents computational (monadic) alternative of let binding.

We first introduce the syntax and mapping to the underlying operations, but both are made precise later §3. To show the breadth of applications, we look at five examples arising from different abstract computations.

2.1 Monadic asynchronous workflows

Asynchronous workflows [99] allow writing non-blocking I/O using a mechanism based on the *continuation monad* (with error handling etc.) The following example shows F# version with an equivalent C# code using single-purpose feature:

```
let getLength url = async {          async Task⟨string⟩ GetLength(string url) {
  let! html = fetchAsync url           var html = await FetchAsync(url);
  do! Async.Sleep 1000                 await Task.Delay(1000);
  return html.Length                   return html.Length;
}                                    }
```

[4] The focus of this paper is *not* on computation expressions, but on their relation to well-known abstractions. Readers unfamiliar with F# may find extended explanation of the mechanism in previous publications [99,9].

Both functions return a computation that expects a *continuation* and then downloads a given URL, waits one second and passes content length to the continuation. The C# version uses the built-in await keyword to represent non-blocking waiting. In F#, the computation is enclosed in the async {...} block, where async is an identifier that refers to the computation builder.

Depending on the operations provided by the builder, different pre-defined keywords are allowed in the computation block. The previous snippet uses let! which represents (monadic) composition and requires the *Bind* operation. This operation also enables the do! keyword which is equivalent to using let! on an unit-returning computation. Finally, the return keyword is mapped to the *Return* operation, so the previous F# snippet is translated as follows:

```
async.Bind(fetchAsync(url), fun html →
    async.Bind(Async.Sleep 1000, fun () →
        async.Return(html.Length)))
```

The two operations form a monad and have the standard types. Assuming $A\tau$ is a type of asynchronous computations, the *Return* has a type $\alpha \rightarrow A\alpha$ and the required type of *Bind* is $A\alpha \rightarrow (\alpha \rightarrow A\beta) \rightarrow A\beta$ (as a convention, we use α, β for universally qualified type variables and τ as for concrete types).

Sequencing and effects. Primitive effectful expressions in F# return unit. Assuming e_1 returns unit, we can sequence expression using $e_1; e_2$ and we can also write effectful if condition without the else clause (which implicitly returns the unit value in the false case). Both of these constructs have their equivalent in the computation expression syntax:

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

If *delay* is true, the workflow waits one second before downloading page and returning it. For monads, it is possible to translate the snippet above using just *Bind* and *Return*, but this approach does not work for other computations §2.2. For this reason, F# requires additional operations – *Zero* represents monadic unit value, *Combine* corresponds to the ";" operator and *Delay* takes an effectful computation and embeds the effects in a (delayed) computation.

Finally, the return! keyword is used to return the result of a computation. It requires an operation *ReturnFrom* of type $A\alpha \rightarrow A\alpha$. This is typically implemented as an identity function – its main purpose is to enable the return! keyword in the syntax, as this may not be alway desirable §2.2.

```
async.Delay(fun () → async.Combine(
    (  if delay then async.Bind(Async.Sleep(1000), fun () → async.Zero())
       else async.Zero() ),
    async.Delay(fun() →
      printfn "Starting..."
      async.ReturnFrom(asyncFetch(url)))))
```

The *Zero* operation has a type $\mathsf{unit} \to A\,\mathsf{unit}$. It is inserted when a computation does not return a value – here, in both branches of the conditional. The result of conditional is composed with the rest of the computation using *Combine* which has a type $A\,\mathsf{unit} \to A\alpha \to A\alpha$. The first argument is a unit-returning computation, which mirrors the ";" operator – the overall computation runs the left-hand side and then returns the result of the right-hand side.

Finally, the *Delay* operation (of type $(\mathsf{unit} \to A\tau) \to A\tau$ is used to wrap any effectful computations (like printing) in the monadic computation to avoid evaluating them before the first part of sequential computation is run.

2.2 Additive parsers and list comprehensions

An asynchronous workflow returns only *one* value, but parsers or list comprehensions may return multiple values. Such computations can be structured using additive monads (MonadPlus in Haskell). These abstractions can be used with F# computation expressions, but they require different typing of *Zero* and *Combine*. It may be also desirable to use different syntax.

Monadic parsers. For monadic parsers, we use a notation similar to the one used in asynchronous workflows. The difference is that we can now use return and return! repeatedly. The following parsers recognize one or more and zero or more repetitions of a given predicate, respectively:

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

The oneOrMore function uses just the monadic interface and so its translation uses *Bind* and *Return*. The zeroOrMore function is more interesting – it combines a parser that returns one or more occurrences with a parser that always succeeds and returns an empty list. This is achieved using the *Combine* operation:

```
let rec zeroOrMore p = parse.Delay(fun () →
  parse.Combine( parse.ReturnFrom(oneOrMore p),
                 parse.Delay(fun() → parse.Return( [] ) )))
```

The *Combine* operation represents the monoidal operation on parsers (either left-biassed or non-deterministic choice) and it has a type $P\alpha \to P\alpha \to P\alpha$. Accordingly, the *Zero* operations is the unit of the monoid. It represents a parser that always fails (returning no values of type α) and has a type $\mathsf{unit} \to P\alpha$.

For effectful sequencing of monads, it only makes sense to use unit-returning values in the left hand side of *Combine* and as the result of *Zero*. However, if a computation supports the monoidal interface, these operations can combine multiple returned values. This shows that the computation expression mechanism needs certain flexibility – although the translation is the same in both cases, the typing needs to depend on the user-defined types of the operations.

List comprehensions. Although list comprehensions implement the same abstract type as parsers, we need to use different syntax if we want to make the syntactic sugar comparable to built-in features in other languages. The following shows F# list comprehension and Python generator side-by-side:

```
let duplicate(list) = seq {          def duplicate(list) :
  for n in list do                     for n in list :
    yield n                              yield n
    yield n * 10 }                       yield n * 10
```

The computations look very similar – they iterate over a source list and produce two results for each input. In contrast, Haskell monad comprehensions [19] allow us to write $[\ n * 10 \mid n \leftarrow \mathit{list}\ ]$ to multiply all elements by 10, but they are not expressive enough to capture duplication. To do that, the code needs to use the monoidal operation (mplus), but that cannot be done inside comprehensions.

Although the F# syntax looks different to what we have seen so far, it is actually very similar. The for and yield constructs are translated to *For* and *Yield* operations which have the same form as *Bind* and *Return*, but provide backing for a different syntax. The translation looks as follows:

```
seq.Delay(fun () → seq.For(list, fun () →
  seq.Combine(seq.Yield(n), seq.Delay(fun () → seq.Yield(n * 10))) ))
```

The *Combine* operation concatenates multiple results and has the standard monoidal type $[\alpha] \to [\alpha] \to [\alpha]$. The type of *For* is that of monadic binding $[\alpha] \to (\alpha \to [\beta]) \to [\beta]$ and *Yield* has a type of monadic unit $\alpha \to [\alpha]$. We could have provided the *Bind* and *Return* operations in the seq builder instead, but this leads to a less intuitive syntax that requires users to write let! for iteration and return for yielding.

As the Python comparison shows, the flexibility of computation expressions means that they are often close to built-in language features. The author of a concrete computation (parse, seq, async, . . .) decides what syntax is appropriate. We can only provide anecdotal recommendation – for computations where the *monoidal* interface is more important, the for/yield notation fits better, while for computations where the *monadic* interface dominates we prefer let! and return.

2.3 Layered asynchronous sequences

It is often useful to combine non-standard aspects of multiple computation types. Abstractly, this has be described using monad transformers [99]. F# does not support monad transformers directly, but they provide a useful conceptual framework. For example, we migth combine non-blocking execution of asynchronous workflows with the ability to return multiple results in list comprehensions – a file download can then produce data in 1kB buffers as they become available. Such computation is captured by *asynchronous sequences* [14].

Assuming $\mathsf{Async}\,\tau$ is the type of asynchronous workflows, the composed computation can be expressed as follows (inspired by the list transformer [99]):

$$\begin{array}{lcl}\textbf{type}\ \mathsf{AsyncSeqInner}\,\tau & = & \mathsf{AsyncNil} \mid \mathsf{AsyncCons}\ \textbf{of}\ \tau \times \mathsf{Async}\,\tau \\ \textbf{type}\ \mathsf{AsyncSeq}\,\tau & = & \mathsf{Async}\,(\mathsf{AsyncSeqInner}\,\tau)\end{array}$$

When provided with a continuation, asynchronous sequence calls it with either AsyncNil (to denote the end of the sequence) or with AsyncCons that carries a value, together with the rest of the asynchronous sequence. It turns out that the flexibility of computation expression makes it possible to provide an elegant syntax for writing computations of this type:

```
let rec urlPerSecond n = asyncSeq {
  do! Async.Sleep 1000
  yield getUrl i
  yield! iterate (i + 1) }
```

```
let pagePerSecond urls = asyncSeq {
  for url in urlPerSecond 0 do
    let! html = asyncFetch url
    yield url, html }
```

The urlPerSecond function creates an asynchronous sequence that produces one URL per second. It uses bind (do!) of the asynchronous workflow monad to wait one second and then composition of asynchronous sequences, together with yield to produce the next URL. The pagePerSecond function uses for to iterate over (bind on) an asynchronous sequence and then let! to wait for (bind on) an asynchronous workflow. The for loop is asynchronous and lazy – it is run each time the caller asks for the next result.

Asynchronous sequences form a monad and so we could use the standard notation for monads with just let! and return. We would then need explicit lifting function that turns an asynchronous workflow into an asynchronous sequence that returns a single value. However, F# computation expressions allow us to do better. We can define both For and Bind with the following types:

$$\begin{array}{rcl}\mathsf{asyncSeq.For} & : & \mathsf{AsyncSeq}\,\alpha \to (\alpha \to \mathsf{AsyncSeq}\,\beta) \to \mathsf{AsyncSeq}\,\beta \\ \mathsf{asyncSeq.Bind} & : & \mathsf{Async}\,\alpha \quad\;\; \to (\alpha \to \mathsf{AsyncSeq}\,\beta) \to \mathsf{AsyncSeq}\,\beta\end{array}$$

We omit the translation of the above example – it is a straightforward variation on what we have seen so far. A more important point is that we can again benefit from the fact that operations of the computation builder are not restricted to a specific type (such as *Bind* for some monad M).

As previously, the choice of the syntax is left to the author of the computation. Here, asynchronous sequences are an additive monad and so we use for/yield. Underlying asynchronous workflows are just monads, so it makes sense to add let! that automatically lifts a workflow to an asynchronous sequence.

An important aspect of realization that asynchronous sequences can be described using a monad transformer means that certain laws hold. In §5.1 we show how these map to the computation expression syntax.

2.4 Applicative formlets

Our last example uses a computation based on *applicative functors* [2], which is a weaker (and thus more common) abstraction than monads. The difference between applicative and monadic computations is that monadic computation can perform different effects depending on values obtained earlier during the computation. On the other hand, the structure of effects of applicative computation is fully determined by its structure.

In other words, it is not possible to choose which computation to run (using let! or do!) based on values obtained in previous let! bindings. The following example demonstrates this using a web form abstraction called formlets [99]:

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

The computation describes two aspects – the rendering and the processing of entered values. The rendering phase uses the fixed structure to produce HTML with text-box and drop-down elements. 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 the form needs to be known without having access to specific values. The syntax uses parallel binding (**let!**. . . **and**. . .), which binds a fixed number of independent computations. The rest of the computation cannot contain other (applicative) bindings.

There are two equivalent ways of defining applicative functors. We use the less common style which uses two operations. *Merge* of type $F\alpha \to F\beta \to F(\alpha, \beta)$ represents composition of structure (without any knowledge of specific values) and *Map* of type $F\alpha \to (\alpha \to \beta) \to F\beta$ transforms the (pure) value. The computation expression from the previous example is translated as follows:

```
formlet.Map
  ( formlet.Merge(Formlet.textBox, Formlet.dropDown ["Male"; "Female"]),
    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 HTML. The rest of the computation is turned into a pure function passed to *Map*. Note that the translation allows uses beyond applicative functors. The let!. . . and. . . syntax can be used with monads to write zip comprehensions [99], which are useful for parsing, parallelism and more [99].

Applicative functors were first introduced to support *applicative* programming style where monads are not needed. The *idiom brackets* notation [99] fits that purpose better. We find that computation expressions provide a useful alternative for more complex code and fit better with the impure nature of F#.

3 Semantics of computation expressions

Computation expressions are blocks representing non-standard computations that is, computation 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].

3.1 Syntax

$$\begin{array}{lll}
\mathit{expr} & = \mathit{expr}\ \{\ \mathit{cexpr}\ \} & \text{(computation expression)} \\
\mathit{cexpr} & = \textbf{let}\ v = \mathit{expr}\ \textbf{in}\ \mathit{cexpr} & \text{(binding value)} \\
 & \mid\ \textbf{let!}\ v = \mathit{expr}\ \textbf{in}\ \mathit{cexpr} & \text{(binding computation)} \\
 & \mid\ \textbf{let!}\ v_1 = \mathit{expr}_1\ \textbf{and}\ \ldots\ \textbf{and}\ v_n = \mathit{expr}_n\ \textbf{in}\ \mathit{cexpr} & \text{(parallel computation binding)} \\
 & \mid\ \textbf{for}\ v\ \textbf{in}\ \mathit{expr}\ \textbf{do}\ \mathit{cexpr} & \text{(for loop computation)} \\
 & \mid\ \textbf{return}\ \mathit{expr} & \text{(return value)} \\
 & \mid\ \textbf{return!}\ \mathit{expr} & \text{(return computation)} \\
 & \mid\ \textbf{yield}\ \mathit{expr} & \text{(yield value)} \\
 & \mid\ \textbf{yield!}\ \mathit{expr} & \text{(yield computation)} \\
 & \mid\ \mathit{cexpr}_1; \mathit{cexpr}_2 & \text{(compose computations)} \\
 & \mid\ \mathit{expr} & \text{(effectful expression)}
\end{array}$$

3.2 Typing

Typing of yield is similar

$$\frac{\Gamma \vdash \mathit{expr} : \sigma \qquad \Gamma \rhd_\sigma \mathit{cexpr} : M\tau}{\Gamma \vdash \mathit{expr}\ \{\ \mathit{cexpr}\ \} : N\tau} \qquad (\sigma.\mathit{Run}\ :\ M\alpha \to N\alpha)$$

$$\frac{\Gamma \vdash \mathit{expr} : \tau_1 \qquad \Gamma, v : \tau_1 \rhd_\sigma \mathit{cexpr} : M\tau_2}{\Gamma \rhd_\sigma \textbf{let}\ v = \mathit{expr}\ \textbf{in}\ \mathit{cexpr} : M\tau_2}$$

$$\frac{\Gamma \vdash \mathit{expr} : M\tau_1 \qquad \Gamma, v : \tau_1 \rhd_\sigma \mathit{cexpr} : N\tau_2}{\Gamma \rhd_\sigma \textbf{let!}\ v = \mathit{expr}\ \textbf{in}\ \mathit{cexpr} : N\tau_2} \qquad (\sigma.\mathit{Bind}\ :\ M\alpha \to (\alpha \to N\beta) \to N\beta)$$

$$\frac{\Gamma \vdash \mathit{expr} : M\tau_1 \qquad \Gamma, v : \tau_1 \rhd_\sigma \mathit{cexpr} : N\tau_2}{\Gamma \rhd_\sigma \textbf{for}\ v\ \textbf{in}\ \mathit{expr}\ \textbf{do}\ \mathit{cexpr} : N\tau_2} \qquad (\sigma.\mathit{For}\ :\ M\alpha \to (\alpha \to N\beta) \to N\beta)$$

$$\frac{\Gamma \vdash \mathit{expr} : \tau}{\Gamma \rhd_\sigma \textbf{return}\ \mathit{expr} : M\tau} \qquad (\sigma.\mathit{Return}\ :\ \alpha \to M\alpha)$$

$$\frac{\Gamma \vdash \mathit{expr} : M\tau}{\Gamma \rhd_\sigma \textbf{return!}\ \mathit{expr} : N\tau} \qquad (\sigma.\mathit{ReturnFrom}\ :\ M\alpha \to N\alpha)$$

$$\frac{\Gamma \rhd_\sigma \mathit{cexpr}_1 : M\tau_1 \qquad \Gamma \rhd_\sigma \mathit{cexpr}_2 : N\tau_2}{\Gamma \rhd_\sigma \mathit{cexpr}_1; \mathit{cexpr}_2 : L\tau_1} \qquad \begin{pmatrix} \sigma.\mathit{Delay} & : (\mathsf{unit} \to N\alpha) \to D\alpha \\ \sigma.\mathit{Combine} & : M\tau_1 \to D\tau_2 \to L\tau_2 \end{pmatrix}$$

$$\frac{\Gamma \vdash \mathit{expr} : \mathsf{unit}}{\Gamma \rhd_\sigma \mathit{expr} : M\tau} \qquad (\sigma.\mathit{Zero}\ :\ \mathsf{unit} \to M\tau)$$

When we write α and β, we assume universal quantification. When we write τ, we mean any instantiation (but the operation may not be universally qualified).

For example, *zero* may have a type M unit or $M\alpha$.

Typing of yield and yield! is similar to the typing of return and return!, so we omit them.

Zero may

3.3 Translation

4 Delayed computations

5 Abstract computation types

5.1 Monad transformers

6 Applicative computations

7 Conclusions

Acknowledgements

A Bonus