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, §2.4, list comprehensions §2.2, monadic parsers §2.3 and formlets for web programming §2.5.

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*. The computation builder is a record of operations that define the computation. It also defines what syntax is available in the block.

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.

2.1 Monadic asynchronous workflows

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

```
async Task<string> GetLength(string url) {
  var html = await FetchAsync(url);
  await Task.Delay(1000);
  return html.Length;
}
```

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

Effects

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

a

```
async.Run(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)) )))
```

imperative monads with sequencing and effects

2.2 Additive sequence expressions

2.3 Additivie parser combinators

2.4 Layered asynchronous sequences

2.5 Applicative formlets

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].

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

```
duplicate list =
  do n ← list
     (return n) `mplus`
       (return$)
```

```
duplicate list =
  do a <- list
     (return a) `mplus` (return $ a * 10)
```

3.1 Syntax

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

3.2 Typing

Typing of yield is similar

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

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

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

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

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

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

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

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

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