Computer Laboratory University of Cambridge

Syntax Matters: Writing abstract computations in F#

Tomas Petricek
University of Cambridge, tp322@cam.ac.uk

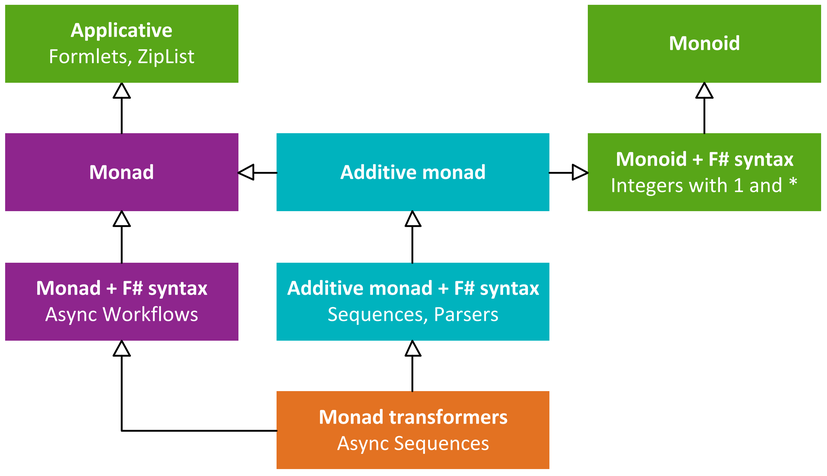
Don Syme
Microsoft Research Cambridge, dsyme@microsoft.com

Introduction

Many programming languages provide syntax that allows writing computations for generating sequences, asynchronous programming or for working with monads. They all use different syntax and work with different abstract computation types.

F# *computation expressions* are a flexible syntactic sugar for writing abstract computations. The library author controls what constructs to use by providing different operations. As a result, they can choose natural syntax for every computation type.

We identify what abstract computations can be encoded using this mechanism and give examples of the most suitable syntax.



Sequence expressions

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

Combines monadic and monoidal computation type

- **combine** : $\text{Seq}\ a \rightarrow \text{Seq}\ a \rightarrow \text{Seq}\ a$
- **yield** : $a \rightarrow \text{Seq}\ a$
- **for** : $\text{Seq}\ a \rightarrow (a \rightarrow \text{Seq}\ b) \rightarrow \text{Seq}\ b$

Asynchronous workflows

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

Monad with imperative control flow constructs

- **bind** : $\text{Async}\ a \rightarrow (a \rightarrow \text{Async}\ b) \rightarrow \text{Async}\ b$
- **for** : $[a] \rightarrow (a \rightarrow \text{Async}\ 1) \rightarrow \text{Async}\ 1$
- **while** : $(1 \rightarrow bool) \rightarrow \text{Async}\ 1 \rightarrow \text{Async}\ 1$

Computation expression design principles

Unify single-purpose syntactic sugar Customize binding and control flow	**Unify extensions**
Reuse the standard F# syntax The library author specifies the syntax	**Standard syntax**
Reinterpret standard F# constructs Make operation types flexible	**Flexible interpretation**

Asynchronous sequences

```
let htmlStrings = asyncSeq {
  for url in addressStream do
    let! html = wc.AsyncDownloadString(url)
    do! Async.Sleep(1000)
    yield url, html }
```

Monad with imperative control flow constructs

- **bind** : $\text{Async}\ a \rightarrow (a \rightarrow \text{AsyncSeq}\ b) \rightarrow \text{AsyncSeq}\ b$
- **for** : $[a] \rightarrow (a \rightarrow \text{AsyncSeq}\ b) \rightarrow \text{AsyncSeq}\ b$
- **for** : $\text{AsyncSeq}\ a \rightarrow (a \rightarrow \text{AsyncSeq}\ b) \rightarrow \text{AsyncSeq}\ b$
- **yield** : $a \rightarrow \text{AsyncSeq}\ a$

Non-standard computations in C# and Python

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

Async in C# 5 (left): Binding using `await` does not block the running thread and uses continuation passing style.

Generators in Python (right). The `yield` keyword is used to return a sequence of results from a function.

Haskell do notation. Syntax for working with monads.

```
def duplicate(inputs):
  for number in inputs:
    yield number
    yield number * 10
```

UNIVERSITY OF CAMBRIDGE