Syntax Matters:
Writing abstract computations in F#

Tomas Petricek
University of Cambridge

Don Syme
Microsoft Research
Non-standard computations in various languages and F#
async Task<string> GetLength(string url) {
    var html = await DownloadAsync(url);
    return html.Length;
}

getLength : string → Async<int>

let getLength url = async {
    let! html = downloadAsync url
    return html.Length
}
Python generators

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

duplicate : seq<int> → seq<int>

let duplicate inputs = seq {
    for number in inputs do
        yield number
        yield number * 10 }
```

Not just a monad!
Unifying single-purpose syntax with F# computation expressions
F# computation expressions

Unify single-purpose extensions
  Custom binding
  Custom returning or yielding
  Custom loops and exceptions

Computation expression principles
  Unify single-purpose syntax
  Reuse standard syntax of F#
  Allow flexible custom interpretation
What **types of computations**?

**Library author** decides

Adding operations enables constructs
Flexible types of operations

Enable custom for loops

```
for : [α] → (α → Mnd β) → Mnd β
for : Seq α → (α → Mnd β) → Mnd β
```

**Paper identifies** **common abstractions**
Finds the most **intuitive syntax**
What can we express?

- **Applicative**
  - Formlets, ZipList

- **Monad**
  - **Applicative**
  - **Formlets, ZipList**
  - **Monad + F# syntax**
  - Async Workflows

- **Additive monad**
  - **Additive monad + F# syntax**
  - Sequences, Parsers

- **Monoid**
  - Monoid + F# syntax
  - Integers with 1 and *

- **Monad transformers**
  - Async Sequences
Common computation types and syntax for them
What can we express?

- **Applicative** (Formlets, ZipList)
- **Monad**
  - **Monad**
  - **Monad** + F# syntax
    - Async Workflows
- **Additive monad**
  - **Additive monad**
  - **Additive monad** + F# syntax
    - Sequences, Parsers
- **Monoid**
  - **Monoid** + F# syntax
    - Integers with 1 and *
- **Monad transformers**
  - **Monad transformers**
    - Async Sequences
Asynchronous and sequences

List transformer applied to async monad

```
let pages = asyncSeq {
    for url in addressStream do
        let! html = wc.AsyncDownload(url)
        yield url, html
}
```

\[
\begin{align*}
\text{for} & : \text{AsyncSeq } \alpha \rightarrow (\alpha \rightarrow \text{AsyncSeq } \beta) \rightarrow \text{AsyncSeq } \beta \\
\text{bind} & : \text{Async } \alpha \rightarrow (\alpha \rightarrow \text{AsyncSeq } \beta) \rightarrow \text{AsyncSeq } \beta \\
\text{for} & : [\alpha] \rightarrow (\alpha \rightarrow \text{AsyncSeq } \beta) \rightarrow \text{AsyncSeq } \beta
\end{align*}
\]
Summary

Syntax matters!
  Reinterpretation of standard syntax
  Better intuition than combinators

Flexibility is good!
  Wide range of computations
  Intuitive syntax for a computation
What can we express?

- **Applicative** (Formlets, ZipList)
- **Monad**
  - **Monad + F# syntax**
    - **Async Workflows**
  - **Additive monad**
  - **Additive monad + F# syntax**
    - **Sequences, Parsers**
- **Monoid**
  - **Monoid + F# syntax**
    - **Integers with 1 and * **
- **Monad transformers**
  - **Async Sequences**
Applicative formlets

Generalization of monad
Computation structure cannot depend on values

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

map : Formlet α → (α → β) → Formlet β
merge : Formlet α → Formlet β → Formlet (α × β)
return : α → Formlet α
What can we express?

- **Applicative**
  - Formlets, ZipList

- **Monad**
  - **Additive monad**
    - **Additive monad + F# syntax**
      - Async Workflows
    - **Monad transformers**
      - Async Sequences

- **Monoid**
  - **Monoid + F# syntax**
    - Integers with 1 and *

Async Workflows
Monad transformers
Async Sequences
Additive monad for sequences

Combines monad and monoid
Uses `for` for monadic binding

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

`for` : `Seq α → (α → Seq β) → Seq β`

`combine` : `Seq α → Seq α → Seq α`

`yield` : `α → Seq α`
Additive monad for parsers

Combines monad and monoid
Uses let! for monadic binding

let rec some p = parse {
  let! x = p
  let! xs = many p
  return x::xs }

and many p = parse {
  return! some p
  return [] }

bind : Seq α → (α → Seq β) → Seq β
combine : Seq α → Seq α → Seq α
return : α → Seq α
What can we express?

- **Applicative**
  - Formlets, ZipList

- **Monad**

- **Monad + F# syntax**
  - Async Workflows

- **Additive monad**

- **Additive monad + F# syntax**
  - Sequences, Parsers

- **Monoid**

- **Monoid + F# syntax**
  - Integers with 1 and *

- **Monad transformers**
  - Async Sequences
Monadic async workflows

Monad with standard F# control flow

async {
    while true do
    for color in [green; orange; red] do
displayLight color }

bind : Async α → (α → Async β) → Async β
for  : [α]       → (α → Async 1) → Async 1
while : (1 → bool) → Async 1 → Async 1