What did not fit in the paper

Monadic exception handling

Monadic exception handling. In an impure functional language that supports exceptions, it is important to provide a mechanism for exception handling within the monadic syntax. The tran­slation restruc­tures code into multiple scopes (by inserting functions), so manual exception handling would require wrapping every sub-expres­sion with a try .. with block.

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

tryWith :

tryFinally :

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. This function is not monadic, which is further discussed in Section 7.

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

Handling of side-effects in monoids

The handling of effects complicates the simple semigroup structure slightly. To be fully precise, the associativity law of semigroups that involve effects can be written as follows (assuming that *ea*, *eb* and *ec* are three expressions, possibly with effects):

Finally, it is worth noting that the *combine* operation used here for semigroups has the same name as *combine* used for sequencing of monads, but it has subtly different type. Sequencing for monads excepts that the first computation returns unit (type ) and the value is ignored while *combine* for semigroups combines the values using the semigroup structure (and so both arguments return a value and , respectively).

More stuff on sequence expressions

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. The monoidal interface provi­des *combine* for concatenating collections and *zero* as the empty collection. Monadic structure provides *bind*, which concatenates all generated collections and *return* for creating a singleton sequence.

Sequences are generated using the yield keyword, which reflects the fact that the computation generates multiple elements. Moreover, the *bind* operation that would normally be defined for sequences overlaps with the *for* operation, which also takes list as the first argument; for sequences, :

for :

bind :

In other words, the *for* operation represents an alternative form of monadic binding that has been always specialized to take as the input. The next section gives more details on this view. For sequences, we choose to define *for* and avoid the let! syntax for binding. The other definitions are the same as in the previous section. In the syntax, this means that iteration over lists can be written using a normal for loop:

**let rec** listFiles dir = seq {

**yield!** Directory.GetFiles(dir)

**for** subdir **in** Directory.GetDirectories(dir) **do**

**yield!** listFiles subdir }

The body of the (recursive) function combines all files generated from the current di­rectory with a sequence that is generated by concatenating (using for) all recursively generated files for all sub-directories of the current folder.

Applicative functor syntax

With my extension, it is possible to define:

merge :

map :

return :

And then we can use

formlet { let! name = Formlet.textBox

and surname = Formlet.textBox

return name + " " + surname }

Alternative class and joinads

When we then add:

choice :

We can write:

formlet { match! loginForm, contactForm with

| login, ? -> return finishOrder login

| ?, info -> return finishUsingTemporary info }

Which then can be further extended to joinads.

Syntactic scope

The *run* operation works as a delimiter (related to delimited continuations?). To implement the imperative computation builder, we define the members below (the values ma do not appear in code. Da=1->ma. Ma is essentially option).

return :

zero :

combine :

delay :

run :

Then we can write:

**let** exists f inp = imperative {

**for** v **in** inp **do**

**if** f v **then return true**

**return false** }

More background

[Important things to note in the rest of the text]

* Translation differs based on the defined operations
* We work with values *ma* representing computations
* Note on notation & desugaring & online code with full translations etc.