Syntax Matters: Writing abstract computations in F#

Tomas Petricek1, Don Syme2

1 University of Cambridge, Cambridge, United Kingdom

2 Microsoft Research, Cambridge, United Kingdom

tp322@cam.ac.uk, dsyme@microsoft.com

**Abstract.** The academic literature describes a number of abstract computation ty­pes such as monads, applicative functors and their compositions. These can be used to describe features of mainstream languages such as generators in Python or asynchronous computa­tions in C# 5, but working with abstract computations without a convenient syntactic sugar is difficult.

In this paper, we describe *computation expressions*, which is a syntactic sugar for working with abstract computations in F# 2.0. Unlike the do notation in Has­kell, computation expressions are not tied to a single kind of abstract com­puta­tions. They support wider range of computations, depending on what opera­tions are available and they also provide greater syntactic flexibility.

As a result, F# programmers are able to use a single syntactic sugar for a wider range of computations including monoidal sequence generators, monadic par­sers and applicative formlets. This removes the need for ad-hoc language exten­sions that provide “nice syntax” for one particular kind of computations.

Introduction

In functional programming, abstract computation types like monads [1] provide a way for composing computations that have some additional behavior. However, monads are not the only example. Applicative functors [2] provide a weaker (and thus more ge­neral) abstraction useful for web programming [18], while MonadPlus [8] is a stronger abstraction used, for example, by parsers [10]. In languages like Haskell, we can write such computations using a mix of combinators, custom operators and syntactic exten­sions like monad comprehensions [19] and do notation.

On the other hand, languages such as Python and C# focus more on the syntax and provide single-purpose features to support asynchrony [20] or sequence generators [11]. Such syntax is not extensible and has limited use, although it can be misused to enco­de other computations [21].

Ideally, we would like to get the best of both worlds. A language should provide a unified syntax that can capture different abstract computations and enable appro­priate parts of the syntax depending on the available operations. In this paper, we expand on the F# language specification [17] and describe F# computation expres­sions in more detail. We focus on covering the wide range of abstract computa­tions that can be captured using a single notation. The main contributions of this paper are:

Practical uses of computation expressions. We demonstrate that F# computation ex­pressions provide a suitable syntax for a wide range of real-world computations and including asynchronous programming (Section 3.1), generators (Section 4) as well as parsers, asynchronous generators (Section 5) and applicative formlets (Section 6)[[1]](#footnote-1).

Encoding abstract computations. We describe abstract computations that can be written using F# computation expressions. Aside from standard computations like mo­nads and monoids (Sections 3 and 4), we show how to encode additive monads, monad transformers (Section 5) and applicative functors (Section 6).

Handling of effects. F# is an impure language, so expressions may have effects. We reflect on how computation expressions embed (untracked) effects in abstract compu­tations. We identify two alternative approaches (Section 3.2), one for monads repre­senting computations and one for monads representing containers.

Computation expressions

The principle that best describes the design of computation expressions is:

*Computation expressions mirror the standard F# syntax in a non-standard setting. They provide non-standard alternatives of syntactic elements that can be used to express computation-specific operations.*

This means that code written as computation expression can use most of the standard F# language features, but they are re-interpreted using the non-standard aspects of the computation. The code can also use extensions to the normal syntax. For example, the let! syntax provides non-standard version of let binding.

## Unifying computations

In this section, we use two examples to demonstrate how F# computation expressions unify language features for writing non-standard computations. We contrast F# with sequence generators in Python and the support for asynchronous programming in C# 5.0, which are both single-purpose extensions.

The following C# 5.0 sample downloads a specified URL without blocking the calling thread. It returns a Task<string> object that can be used to register callback that will be called when the operation completes:

**async** Task<string> GetLength(**string** url) {

**var** html = **await** DownloadAsync(url);

**return** html.Length;

}

The body of the method uses standard C# language features including var for variable binding and return for returning the result. However, it also uses the non-standard con­struct await that specifies that DownloadAsync returns an I/O operation and that the rest of the code should be called when the download completes. Using F# computation expressions, the body can be written as follows:

async { **let!** html = downloadAsync(url)

**return** html.Length }

F# computation expressions are enclosed in a block like async { .. } that determines the meaning. The async identifier is a value referring to a global object. It provides operations that define the meaning of non-standard constructs. We write these opera­tions as functions, but they are actually members of an object.

The snippet uses a non-standard version of a let binding, written as let!, to denote the fact that the download is done asynchronously. This operation is interpreted using the Bind member of the async object. This example closely corresponds to the do nota­tion for monads in Haskell [8], but we will see that we can use many other constructs.

As another example, consider the following Python function that duplicates ele­ments in a list and multiplies the second repetition by 10:

**def** duplicate(list):

**for** value **in** list:

**yield** n

**yield** n \* 10

Monad comprehensions [19] allow us to write [ n \* 10 | n <- list ] to multi­ply all elements by 10, but they are not expressive enough to express duplication. The prob­lem is that the code also needs monoidal structure that concatenates two lists. This can be done using the MonadPlus type class and mplus operation, but then we cannot bene­fit from any syntactic extension. In F#, we can write the sample as follows:

seq { **for** n **in** numbers **do**

**yield** n

**yield** n \* 10 }

The seq { .. } block is again a computation expression, but it uses a different syntax, which is more appropriate for generators. As already noted, this example re­quires mo­noidal structure, which is provided by Combine member of seq. When the member is defined, the computation can produce multiple results using either yield or return.

Monads

Monads, introduced by Moggi [5] and popularized by Wadler [1], are the most widely known class of abstract computations. In purely functional languages, they are useful for propagating state, exceptions or I/O effects [6]. The most prominent monads in F# are *asynchronous workflows* [7]. They are used for writing non-blocking code that involves I/O operations without explicit use of callbacks.

Functional monads

To support monads, the computation must provide the usual *bind* and *return* opera­tions of the types below (we write types as instead of M<a> used in F#):

return :

bind :

The syntax for writing monadic code resembles normal F# code with variable bin­ding. Instead of ordinary let we write let! and returning of a value is not implicit, but requires the use of return keyword. Defining *bind* also enables the do! keyword, which corresponds to do – it is used to call asynchronous operations that return unit.

Using asynchronous workflows as an example, the following code calls function that asynchronously downloads web page (with return type Async<string>), when waits 1 second and returns the length of the page in characters:

async { **let!** html = asyncDownloadUrl url

**do!** Async.Sleep(1000)

**return** html.Length }

The do! notation is treated as a monadic binding that matches the result of computa­tion against the unit pattern and it is equivalent to writing let! () = e. The above snippet is translated using two nested calls to *bind* and a single call to *return*:

async.Bind(asyncDownloadUrl url, **fun** html ->

async.Bind(Async.Sleep(1000), **fun** () ->

async.Return(html.Length)))

The return construct is used to lift a value into a monad, but we also need syntax for returning the result of an existing monadic computation. To keep the syntax uniform, F# uses the return! keyword (let binds a value, let! binds a computation, return returns a value and return! returns a computation). The keyword is only enabled if the computation defines the following member:

returnFrom :

The member is required in order to specify that computation expression should allow the return! syntax. Computations where the syntax is not desirable are shown in Sec­tion 4. Although the operation can have some behavior (it is applied to the retur­ned computation), for monads it should be just an identity function. After adding the member, we can use the following syntax to return the result of a computation:

async { **let!** url = asyncLoadUrl()

**return!** asyncDownloadUrl(url) }

The syntax introduced so far is very similar to the do notation in Haskell [8] with the in Haskell, let! binding is followed by an expression of type *ma*. This means that return! is not needed and return is an ordinary function. In the next section, we look at additional constructs that are allowed in monadic syntax in F#.

Monads with sequencing and effects

The F# language the uses unit type to represent the result of effectful computation and pro­vides the sequencing operator ; for writing them. The expression e1;­ e2 first evalu­ates e1 which is required to return unit, then evaluates e2 and returns its result. In addition, F# also allows if expression with­out the else branch (if e1 then e2), which requires e2 to return unit and implicitly returns unit in the false case.

Monadic versions of sequencing and unit could be expressed just in terms of *bind* and *return*, but that would exclude other uses of the syntax (discussed in Section 5). For this reason, F# requires definition of other operations. There are two alternatives depending on whether the computation *ma* can capture delayed effects or not.

Sequencing for monadic computations. If the value *ma* represents a computation that can be executed later, it is possible to define a *delay* operation that takes an effectful fun­ction and wraps it in an *ma* value without performing the effects. Aside from *delay*, we define the following operations:

delay :

zero :

combine :

run :

All operations other than *delay* have default implementations in terms of *bind* and *return*. We leave these out as an exercise. The meaning of the operations is that *zero* represents monadic unit value and combine corresponds to the ; operator. The *run* operation wraps the entire computation expression (which is delayed using *delay*) and can be implemented as identity function. We will see its utility in the next section.

The additional operations are used to compose computations of forms other than binding (let! and do!) with the translation of consequent constructs. An example that does not require any other operations is the if construct:

async { **let!** html = asyncDownloadUrl url

**if** html <> "" **then**

**do!** Async.Sleep(1000)

**return** html.Length }

The example represents a computation that asynchronously downloads a page and then suspends for 1 second if the page is non-empty before returning its length. The body of the first monadic binding (starting with if) is translated as follows:

async.Run(async.Delay(**fun** () ->

async.Bind(asyncDownloadUrl url, **fun** html ->

async.Combine(

( **if** html <> "" **then**

async.Bind(Async.Sleep(1000), **fun** () -> async.Zero())

**else** async.Zero() ),

async.Delay(**fun** () -> async.Return(html.Length)) ))))

The true branch of if is translated as a binding that returns the *zero* computation. To return value of the same type from both branches, the false branch is added returning *zero* as well. The result is then com­bined with the upcoming *return*, which is wrapped in *delay* to avoid performing effects when evaluating the arguments of *combine*.

The rest of the translation uses an additional *bind* (for the html value), wraps the whole computation in a lambda function passed to *delay* and then applies *run* on the result. The purpose of the overall *delay* is, again, to wrap any im­mediate effects (in the first argument of *bind*) inside the computation.

Note that the first argument of the *combine* operation has to be a unit-returning computation, which means that it is not possible to write C-style return to jump from the middle of a computation. We return to this topic in Section 5.

Sequencing for monadic containers. If the value *ma* represents a wrapped, non-dela­yed value (such as a value of type option<'T>), then we cannot implement a *delay* function of the previous type without performing the effects immediately. However, the translation is only syntactic and so it allows different typing of the operations:

delay :

zero :

combine :

run :

In this alternative, the *delay* is an identity function that returns the function it gets as an argu­ment. The result of *delay* is used as a second argument of combine and it is also passed to *run*, so we modify their types accordingly. The operations *zero* and *combine* can be defined as *return* and *bind*, respectively. Finally, *run* applies the dela­yed function (potentially performing the effects) to get the result of a computation.

This variant makes it possible to sequence computations of monads that do not de­lay the contained value such as the Maybe monad:

maybe { **if** b = 0 **then return!** fail()

**return** a / b }

The translation for this snippet calls combine with translation of the first and second line as the first and second argument, respectively. If *delay* evaluated the function to obtain an *ma* value, it would evaluate a / b even if b equals zero. By returning a fun­ction, the *combine* operation may not need to evaluate the second computation (if the first one fails), avoiding a runtime exception caused by division by zero.

Unifying delayed computations. So far, we used distinct types to represent delayed computations. For monadic computations, we used the type and for monadic containers, we used . We can generalize the two cases by using a different abstract type for delayed computations, which leads to the following signatures:

delay :

combine :

run :

To our knowledge, the only types used for in practice so far are and . However, the generalization simplifies the discussion in upcoming sections and it may allow interesting uses. The computation may itself have a monadic structure, but this is not required. We only require that delaying an effect-free function and eva­luating it using *run* is an identity:

The equation holds for the standard definitions of *delay* and *run* and it guarantees that the added operations do not change the meaning in case when they are not needed.

Monadic control flow constructs

The syntax allowed inside computation expressions aims to provide computation-spe­cific versions of most of the standard control flow constructs of F#. In this section, we look at the support for (imperative) loops and exception handling.

Looping syntax. If a computation builder defines *zero* and *combine*, we can use them to add two additional operations that enable looping constructs inside the computation expression syntax. Similarly to operations defined in the previous section, these can be defined in terms of other operations for monads, but the F# com­putation expres­sion syntax allows other useful definitions (Section 4, 5.2):

for :

while :

The *for* operation represents sequencing of computations generated from a list and *while* represents repeated evaluation of a computation while a condition (rely­ing on mutable state) holds. The first argument is a function that evaluates the condi­tion. The second argument represents a delayed body. In *for*, the second argument is always a function, so *delay* is not used.

In asynchronous workflows, looping constructs are useful for writing repeating and long-running computations. The following example is adapted from [7]:

async { **while true do**

**for** color **in** [green; orange; red] **do**

**do!** Async.Sleep(1000)

displayLight color }

The code creates a workflow that repeatedly changes the color of a semaphore light with a 1 second delay. The function displayLight mutates the user interface.

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*.

Semigroups and monoids

A semigroup is a set with an associative operation . Examples of semigroups in fun­ctional programming include lists (with concatenation), but also option values (with a left-biased operation for combination). Semigroups appear mainly as parts of monoids, which also have a special element e that behaves as identity of the opera­tion (). We start by looking at semigroups to explain how delaying of side-effects is handled. The identity element is added in a later section.

Semigroups. Similarly to the handling of effects monads, encoding of a computation with semi­group structure re­quires a *delay* operation of type , where is typically either for computations or for containers. To use com­puta­tion expression syntax for semigroup, we need to define the following operations:

yield :

combine :

delay :

run :

The *delay* and *run* operations have the same type and the same purpose as previously for monads. The *yield* operation is similar to *return*. However, defining *yield* instead of *return* means that the computation uses yield syntax instead of return. Although this is just a syntactical difference, it hints that the computation is designed to produce multiple results (we return to this topic in Section 5).

The *combine* function represents the binary operation of the semigroup. Because F# is an eager language, the second argument of the *combine* operation needs to be delayed, so that effects that might happen when evaluating it only happen at the time when the computation is required (i.e. when evaluating nth element of a lazy list).

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).

As an example, consider semigroup of Booleans with conjunction[[2]](#footnote-2). The *combine* operation evaluates the second argument only if the value of the first argument is true. This means that the fol­lowing code returns false without printing “calculating”:

conj { **yield false**

printfn *"calculating"*

**yield true** }

The translation applies *de­lay* on the expression that cal­culates the second argument of *combine*. In addition, it also applies *delay* to the entire expression, but the result is then passed to *run*, which performs the evaluation for container-like structures:

conj.Run(conj.Delay(

conj.Combine(conj.Yield(1), conj.Delay(**fun** () ->

printfn "calculating"; conj.Yield(2))) ))

In the previous example, the type cannot capture delayed computation, so *delay* has to return . On the other hand, lazy list (with concatenation) is a computa­tion with semigroup (monoid) structure where delay wraps effects inside values.

Extending semigroups to monoids. Most of the semigroup structures that are used in functi­onal programming are monoids with an additional identity element (true for conjunction and empty list for lists). The identity element can be added by defining the *zero* operation. In order to allow composing computations using existing *ma* values, we also include an operation *yieldFrom*:

yieldFrom :

zero :

The *yieldFrom* operation plays similar role as *returnFrom* from Section 3.1. The only difference is that it enables yield! syntax instead of return! As noted previously, this is just a syntactic choice to make the language more familiar (see also Section 5.

The *zero* operation is similar to *zero* for monads (Section 3.2), but the re­sult type is instead of . This shows that the operation plays a different role – instead of re­presenting computation that returns unit, it represents a computation that does not contain a value (and combining it with other computation has no effect).

If we extend the conj builder from an earlier section with the above two functions, we can write the following code to validate user name and password:

conj { **if** strictPolicy **then yield** validPassword

**yield!** validateUser name }

The example assumes that validateUser is a function with body implemented using conj. When strictPolicy is set to true, the Boolean value validPassword is included in the conjunction, otherwise the computation only considers validation rules in the validateUser function. The translation of the body looks as follows:

conj.Combine

( ( **if** strictPolicy **then** conj.Yield(validPassword)

**else** conj.Zero() ),

conj.Delay(**fun** () -> conj.YieldFrom(validateUser name)) )

Control flow constructs for monoids. Similarly to monads a computation with a mo­noid structure can provide additional members to allow the use of standard F# control flow constructs. If a computation builder defines *zero* and *combine*, we can provide the following operations to allow looping constructs:

for :

while :

The definition in terms of *zero* and *combine* is exactly the same as for monads (Sec­tion 3.3). However, *combine* has a different type for monoids, so the type of *for* and *while* is also different. The second argument (representing the loop body) can now return values other than unit and the values are combined using the binary operation of a monoid. Using the conj computation as an example, the additional members allow writing code that validates all user names in a list as follows:

conj { **for** name **in** userNames **do**

**yield!** validateUser name }

The body passed to *for* in the translated code is a function that calls validateUser and then passes the result to *returnFrom* (which is just an identity function).

Composed computations

The last two sections discussed basic types of computations that can be encoded using F# computation expressions. A unified syntax can be used for writing compu­tations that form monads and monoids. In this section, we take one step further. We look at composed computations with more complex structures.

Additive monads

An additive monad is a monad with a monoid structure. In Haskell, this struc­ture is captured by MonadPlus. Common examples of additive monads are parser combi­nators and collections. Both of the examples can be encoded using F# computation expres­sions, but the desirable syntax differs.

Parser combinators. Monadic parsers [10] provide both monadic and monoidal in­terface. The *unit* operation represents a parser that always succeeds without con­suming any input and *bind* represents sequencing (where the second parser may de­pend on the value parsed first). The monoid structure defines *combine* as either left-biased or non-deterministic choice and *zero* represents a failing parser.

When defining additive monads, we need to choose whether to use the return or the yield keyword. This is a matter of style, but it depends on which interface is con­sidered more important. For parsers, we prefer monad and define *return*:

return :

bind :

zero :

combine :

The type of *bind*, *return* and *zero* is standard. The type of *combine* follows the defini­tion used by monoids (with arguments and ), which is more general than the one used earlier for monads (with as the first argument).

To make the definition com­plete, we also need to define *returnFrom* and *delay* with *run*. Sample parsers that recognize one or more (some) and any number (many) repetitions of a predicate are written as follows:

**let rec** some p = parse {

**let!** x = p

**let!** xs = many p

**return** x::xs }

**and** many p = parse {

**return!** some p

**return** [] }

The definition of some uses monadic features to say that the parser should parse p fol­lowed by zero or more repetitions of p. On the other hand, the definition of many uses monoidal interface to combine two alternatives. The first one parses one or more repetitions of p and the second one succeeds without consuming any input.

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.

Monad transformers

Monad transformers [13] provide a mechanism for building more complex monads from simpler ones. Examples used in Haskell include transformers that add state or exceptions to other monad, which makes it possible to layer aspects of computations. F# does not support higher-kinded types, so monad transformers cannot be encoded directly, but they still provide useful framework for documenting computations.

We look at one practical computation that can be viewed as an appli­cation of a monad transformer. Asynchronous workflows [7], discussed earlier, provide a way for writing non-blocking computations that return a single result. How­ever, asynchronous computations might need to return multiple results. A file download may return the data in 1kB buffers. Asynchronous sequences [14] make this possible by defining a computation that generates results asynchronously. This can be expressed as an application of list monad transformer [15,16] to the asynchronous workflow monad:

**type** ListT<'M, 'T> = 'M<ListTInner<'M, 'T>>

**and** ListTInner<'M, 'T> = Nil | Cons **of** 'T \* ListT<'M, 'T>

**type** AsyncSeq<'T> = ListT<Async, 'T>

This sample is not valid F# (the type parameter 'M would have a kind \* -> \*), but it demonstrates the idea. An asynchronous sequence is a non-blocking computation that eventually produces either Nil (representing the end) or a Cons consisting of a value and an asynchronous workflow that produces the next element. In case of asyn­chronous sequences, we can lift both of the composed types to the AsyncSeq<'T>:

* An asynchronous workflow can be turned into an asynchronous seq­uen­ce that produces a single value (using *return* operation of sequence).
* A list can be turned into an asynchronous seq­­uence that produces values of the list one-by-one immediately (using *return* operation of asynchronous workflows).

This means that *bind* could take a type AsyncSeq<'T> (written as ) and other types, [*a*] for lists and for asynchronous workflows, could be lifted using explicit functi­ons. However, to simplify the syntax, we can define *for* and *bind* and also use ad-hoc polymorphism (overloading) for members in F# to define two variants of *for*:

for :

for :

bind :

The first *for* operation represents the monadic bind for asynchronous sequences. However, asynchronous sequences also provide the monoidal interface (*combine* and *zero*) and so we follow the example of standard sequence expressions (Section 5.1) and use the for syntax. The second *for* defines an overloaded version that iterates over standard list and concatenates the generated asynchronous sequences.

Finally, we also define bind operation, but its first argument is just an asynchro­nous workflow instead of asynchronous sequence. The operation can be implemented by lifting the workflow to a singleton sequence and then using the first overload of *for*, but it allows the users to write let! in the computation syntax and mean exactly the same thing as when using let! in asynchronous workflows.

From a practical per­spective, asynchronous sequences can be viewed as extensions of asynchronous work­flows that replace return with add and allow asynchronous iteration. Assuming that urls is an asynchronous sequence of URLs (perhaps gener­ated by reading inputs from the user), the following snippet asynchronously produces the URLs together with their HTML content:

**let** pages = asyncSeq {

**let** wc = **new** WebClient()

**for** url **in** urls **do**

**let!** html = wc.AsyncDownloadString(Uri(url))

**yield** url, html }

Assuming that urls is an asynchronous sequence that produces URLs, the computa­tion first creates WebClient object, then it asks the urls computation to generate the first URL, downloads the content and returns that to the caller. When the caller re­quests the next value, the computation runs the next iteration of the for loop by get­ting the next URL, downloading it and yielding the result.

The use of the for keyword for monadic bind on asynchronous sequences and let! for binding on asynchronous workflows makes it easy to distinguish between binding that runs the following expression just once and binding that evaluates the following expression repeatedly. This matches the expectations of the user – the body of for loop is executed repeatedly, while the body of let binding is executed just once.

Applicative functors

Applicative functors [2] provide a weaker abstraction than mo­nads – every monad is an applicative functor, but not conversely. For example, formlets [18] are an appli­cative functor for composing HTML forms that are not monads. Using a weaker abstrac­tion also allows more ef­ficient implementation of parsers [25].

There are two ways of defining applicative functors. The first relies on an opera­tion of type and is more suitable for writing computations in an applicative style and propagating their effects. We use the second style, which is more suitable for computational interpretation used, for example, by formlets:

merge :

map :

return :

The declaration replaces monadic *bind* with *map* and *merge*; *map* ap­plies function to values carried by the abstract computation and *merge* composes additional aspects (or effects) of two computations and combines their va­lues using a tuple.

The difference between applicative functors and monads is that monadic compu­tations can perform different effects depending on values obtained as the result. On the other hand, the additional (effect) structure of applicative functors is fully determi­ned regardless of the calculations performed using *map*.

The next sample uses formlets to create a registration form consisting of a text­box and a dropdown. When the values are entered, it produces a string with user details:

**let** userFormlet = formlet {

**let!** name = Formlet.textBox

**and** gender = Formlet.dropDown ["Male"; "Female"]

**return** name + " " + gender }

The computation describes two aspect of HTML form – the rendering and the proces­sing of entered values. The rendering phase uses the fixed structure of the computa­tion to produce HTML code representing the form. 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 applicative computations cannot depends on the values, so the syntax for uses parallel binding (let! .. and ..), which binds a fixed number of in­dependent computations. The rest of the compu­tation cannot contain other bindings. The computation expression from the previous example is translated as follows:

formlet.Map

( formlet.Merge Formlet.textBox (Formlet.dropDown [ ... ]),

**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. The rest of the computation is transla­ted into projection (by removing the return keyword) and is applied to the composed computation (formlet) using *map*. In formlets, the projection function is evaluated only in the input processing phase.

Related work

Applicative, monads and arrows. There are syntactic extensions for a number of abstract computation types. Haskell provides monad comprehensions [19] and do notation for working with monads; McBride [2] proposes a notation for applicative fun­ctors. However, these notations are all quite different – in this paper, we tried to provide a single syntax that can capture a wider range of computations. Our syntax is not sufficient to encode arrow notation [27], which is an interesting future problem.

Delimited continuations. Filinski demonstrated [26] that monadic computations can be encoded using continuations. An intriguing question is whether this work could lead to a simpler notation for monads and more complex structures discussed in this paper. This alternative could be attractive for languages that support delimited conti­nuations like Scala [9]. The reset operation of delimited continuations seems related to the *run* function of computation expressions, which can be also used to restrict the scope of behavior. This aspect is used by the imperative computation expression [12].

Generators and monads. Generators, also called iterators [23] are the most common class of non-standard computations in main-stream languages. Although they are designed specifically for collections, they have been used to encode other monadic computations [21] and also delimited continuations [24]. However, the fact that the syntax has been designed for another purpose is a limiting factor.

Languages with effects. As far as we are aware, the handling of effects allowed by a host language in monadic computations has not been discussed previously. However, our *delay* operation is similar to Filinski’s *reify* operation [3]. Instead of capturing all ef­fects, it combines effects of the function arrow and effects associated with the resul­ting computation. Using a precise type system with annota­tions for effects, such as [4], the type of *delay* would be written as:

The operation represents redistribution of effects. Assuming that represents a com­bina­tion of effects, it must hold that . A default implementation of *delay* simply applies the function (giving ), but for computations that can cap­ture effects, it is desirable to provide operation where .

Conclusions

We presented F# computation expressions, which provide a unified syntactic sugar for working with a wide range of abstract computations. We showed that a single syntactic mechanism can be used for working with monoids, monads and applicative functors, as well as computations composed using monad transformers. We believe that an easy to use syntax is the key for bringing these abstractions into practice.

References

1. Wadler, P.: Monads for functional programming. In LNCS Vol. 925, 1995.

2. McBride, C. and Paterson, R.: Applicative programming with effects, Journal of Func. Programming 18 (2008)

3. Filinski, A.: Monads in Action. In Proceedings of POPL 2012.

4. Wadler, P. and Thiemann, P.: The Marriage of Effects and Monads. In ACM Trans. Comput. Logic, vol. 4, num. 1, pp. 1-32, January 2003.

5. Moggi, E.: Notions of Computation and Monads. In Inf. Comput., vol 93, pp. 55-92, 1991

6. Peyton Jones, S. and Wadler, P.: Imperative Functional Programming. POPL, 1993.

7. Syme, D., Petricek, T. and Lomov, D.: The F# Asynchronous Programming Model.   
In Proceedings of PADL 2011.

8. Peyton Jones, S., et al.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, 2003, ISBN: 9780521826143

9. Rompf, T., Maier, I. and Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In Proceedings of ICFP, 2009

10. Hutton, G. and Meijer, E.: Monadic Parsing in Haskell. In J. Funct. Program.,   
vol. 8, num. 4, pp. 437-444, July 1998

11. Hejlsberg, A., Wiltamuth, S., Golde, P.: C# Language Specification, Addison-Wesley, 2003

12. Petricek, T.: Imperative Computations in F# (I. and II.), Unpublished. Retrieved 23 March 2012. Available online at: http://tomasp.net/blog/imperative-ii-break.aspx

13. Liang, S., Hudak, P. and Jones, M.: Monad Transformers and Modular Interpreters. In Proceedings of POPL 1995.

14. Petricek, T.: Programming with F# Asynchronous Sequences, Unpublished. Retrieved 23 March 2012, Available online at: http://tomasp.net/blog/async-sequences.aspx

15. Wadler, P., Taha, W. and MacQueen, D.: How to add laziness to a strict language, without even being odd. In Proceedings of Workshop on Standard ML, September 1998.

16. Haskell Wiki. ListT done right (Unpublished). Retrieved 23 March 2012, Available online at: http://www.haskell.org/haskellwiki/ListT\_done\_right

17. Syme, D. et al.: The F# 2.0 Language Specification (April 2010), Retrieved 23 March 2012, Available online at: http://tinyurl.com/fsharp-spec

18. Cooper, E., Lindley, S., Wadler, P. and Yallop, J.: The Essence of Form Abstraction. In Proceedings of APLAS, 2008.

19. Giorgidze, G., Grust, T., Schweinsberg, N. and Weijers, J.: Bringing Back Monad Comprehensions. In Proceedings of Haskell Symposium, 2011, Tokyo, Japan

20. Bierman, G., Russo, C., Mainland, G., Meijer, E. and Torgersen, M.: Pause 'n' play: Formalizing asynchronous C#. In Proceedings of ECOOP, 2012

21. Petricek, T.: Encoding Monadic Computations in C# using Iterators. In Proceedings of ITAT 2009, Slovakia.

22. Petricek, T.: Variations in F#. Retrieved 24 March 2012, Available online at: http://tomasp.net/blog/fsharp-variations-joinads.aspx

23. Jacobs, B., Meijer, E., Piessens, F. and Schulte, W.: Iterators Revisited. In Proceedings of the 7th ECOOP Workshop on Formal Techniques for Java-like Programs, 2005

24. James, R. P. and Sabry, A.: Yield: Mainstream Delimited Continuations. In Proceedings of Theory and Practice of Delimited Continuations, 2011.

25. Swierstra, S. D.: Combinator Parsing: A Short Tutorial, In Language Engineering and Rigorous Software Development, pp. 252-300, 2009

26. Filinski, A.: Representing layered monads. In Proceedings of POPL, 1999

27. Paterson, R.: A new notation for arrows. In Proceedings of ICFP, 2001

1. In Sections 2, 3 and 4 we describe computation expressions as they were designed by Syme et al. [17] for F# 2.0. Section 6 uses a research extension proposed by Petricek [22]. [↑](#footnote-ref-1)
2. This is a simplified example, but it is based on a computation that encodes compu­tations that provide imperative return and break with continue for loops in F# [12]. [↑](#footnote-ref-2)