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

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 §4.

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 §2.4 and discuss the threatment of effects §3.4 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*, which is a record of operations that define the computation. It also defines what syntax is available in the block[4].

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.

We first introduce the syntax and mapping to the underlying operations, but both are made precise later §3. To show the breadth of applications, we look at five examples arising from different abstract computations.

2.1 Monadic asynchronous workflows

Asynchronous workflows [99] allow writing non-blocking I/O using a mechanism based on the *continuation monad* (with error handling etc.) The following example shows F# version with an equivalent C# code using single-purpose feature:

```
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;
}
```

[4] The focus of this paper is *not* on computation expressions, but on their relation to well-known abstractions. Readers unfamiliar with F# may find extended explanation of the mechanism in previous publications [99,9].

Both functions return a computation that expects a *continuation* and then downloads a given URL, waits one second and passes content length to the continuation. The C# version uses the built-in await keyword to represent non-blocking waiting. In F#, the computation is enclosed in the async $\{\ldots\}$ block, where async is an identifier that refers to the computation builder.

Depending on the operations provided by the builder, different pre-defined keywords are allowed in the computation block. The previous snippet uses let! which represents (monadic) composition and requires the *Bind* operation. This operation also enables the do! keyword which is equivalent to using let! on an unit-returning computation. Finally, the return keyword is mapped to the *Return* operation, so the previous F# snippet is translated as follows:

$$\begin{array}{l}\mathsf{async.Bind}(\mathsf{fetchAsync}(url), \textbf{fun}\ html \rightarrow \\ \quad \mathsf{async.Bind}(\mathsf{Async.Sleep}\ 1000, \textbf{fun}\ () \rightarrow \\ \quad\quad \mathsf{async.Return}(html.\mathsf{Length})))\end{array}$$

The two operations form a monad and have the standard types. Assuming $A\tau$ is a type of asynchronous computations, the *Return* has a type $\alpha \rightarrow A\alpha$ and the required type of *Bind* is $A\alpha \rightarrow (\alpha \rightarrow A\beta) \rightarrow A\beta$ (as a convention, we use α, β for universally qualified type variables and τ as for concrete types) [5].

Sequencing and effects. Primitive effectful expressions in F# return unit. Assuming e_1 returns unit, we can sequence expression using $e_1; e_2$ and we can also write effectful if condition without the else clause (which implicitly returns the unit value in the false case). Both of these constructs have their equivalent in the computation expression syntax:

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

If *delay* is true, the workflow waits one second before downloading page and returning it. For monads, it is possible to translate the snippet above using just *Bind* and *Return*, but this approach does not work for other computations §2.2. For this reason, F# requires additional operations – *Zero* represents monadic unit value, *Combine* corresponds to the ";" operator and *Delay* takes an effectful computation and embeds the effects in a (delayed) computation.

We also use the return! keyword, which returns the result of a computation and requires an operation *ReturnFrom* of type $A\alpha \rightarrow A\alpha$. This is typically implemented as an identity function – its main purpose is to enable the return! keyword in the syntax, as this may not be alway desirable §2.2.

```
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)))))
```

[5] For the purpose of this paper, we write type application using a light notation $T\tau$.

The *Zero* operation has a type $\mathsf{unit} \rightarrow A\,\mathsf{unit}$. It is inserted when a computation does not return a value – here, in both branches of the conditional. The result of conditional is composed with the rest of the computation using *Combine* which has a type $A\,\mathsf{unit} \rightarrow A\alpha \rightarrow A\alpha$. The first argument is a unit-returning computation, which mirrors the ";" operator – the overall computation runs the left-hand side and then returns the result of the right-hand side.

Finally, the *Delay* operation (of type $(\mathsf{unit} \rightarrow A\tau) \rightarrow A\tau$ is used to wrap any effectful computations (like printing) in the monadic computation to avoid evaluating them before the first part of sequential computation is run.

2.2 Additive parsers and list comprehensions

An asynchronous workflow returns only *one* value, but parsers or list comprehensions may return multiple values. Such computations can be structured using additive monads (MonadPlus in Haskell). These abstractions can be used with F# computation expressions, but they require different typing of *Zero* and *Combine*. It may be also desirable to use different syntax.

Monadic parsers. For monadic parsers, we use a notation similar to the one used in asynchronous workflows. The difference is that we can now use return and return! repeatedly. The following parsers recognize one or more and zero or more repetitions of a given predicate, respectively:

```
let rec zeroOrMore p = parse {
  return! oneOrMore p
  return [] }

and oneOrMore p = parse {
  let! x = p
  let! xs = zeroOrMore p
  return x :: xs }
```

The oneOrMore function uses just the monadic interface and so its translation uses *Bind* and *Return*. The zeroOrMore function is more interesting – it combines a parser that returns one or more occurrences with a parser that always succeeds and returns an empty list. This is achieved using the *Combine* operation:

```
let rec zeroOrMore p = parse.Delay(fun () →
  parse.Combine( parse.ReturnFrom(oneOrMore p),
                 parse.Delay(fun() → parse.Return( [] ) )))
```

The *Combine* operation represents the monoidal operation on parsers (either left-biassed or non-deterministic choice) and it has a type $P\alpha \rightarrow P\alpha \rightarrow P\alpha$. Accordingly, the *Zero* operations is the unit of the monoid. It represents a parser that always fails (returning no values of type α) and has a type $\mathsf{unit} \rightarrow P\alpha$.

For effectful sequencing of monads, it only makes sense to use unit-returning values in the left hand side of *Combine* and as the result of *Zero*. However, if a computation supports the monoidal interface, these operations can combine multiple returned values. This shows that the computation expression mechanism needs certain flexibility – although the translation is the same in both cases, the typing needs to depend on the user-defined types of the operations.

List comprehensions. Although list comprehensions implement the same abstract type as parsers, we need to use different syntax if we want to make the syntactic sugar comparable to built-in features in other languages. The following shows F# list comprehension and Python generator side-by-side:

$$\begin{array}{ll}
\textbf{let}\ \mathsf{duplicate}(list) = \mathsf{seq}\ \{ & \qquad \textbf{def}\ \mathsf{duplicate}(list): \\
\quad \textbf{for}\ n\ \textbf{in}\ list\ \textbf{do} & \qquad \quad \textbf{for}\ n\ \textbf{in}\ list: \\
\quad\quad \textbf{yield}\ n & \qquad\quad\quad \textbf{yield}\ n \\
\quad\quad \textbf{yield}\ n * 10\ \} & \qquad\quad\quad \textbf{yield}\ n * 10
\end{array}$$

The computations look very similar – they iterate over a source list and produce two results for each input. In contrast, Haskell monad comprehensions [19] allow us to write $[\ n * 10 \mid n \leftarrow list\ ]$ to multiply all elements by 10, but they are not expressive enough to capture duplication. To do that, the code needs to use the monoidal operation (mplus), but that cannot be done inside comprehensions.

Although the F# syntax looks different to what we have seen so far, it is actually very similar. The for and yield constructs are translated to *For* and *Yield* operations which have the same form as *Bind* and *Return*, but provide backing for a different syntax. The translation looks as follows:

$$\begin{array}{l}
\mathsf{seq.Delay}(\textbf{fun}\ () \rightarrow \mathsf{seq.For}(list, \textbf{fun}\ () \rightarrow \\
\quad \mathsf{seq.Combine}(\mathsf{seq.Yield}(n), \mathsf{seq.Delay}(\textbf{fun}\ () \rightarrow \mathsf{seq.Yield}(n * 10)))\ ))
\end{array}$$

The *Combine* operation concatenates multiple results and has the standard monoidal type $[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$. The type of *For* is that of monadic binding $[\alpha] \rightarrow (\alpha \rightarrow [\beta]) \rightarrow [\beta]$ and *Yield* has a type of monadic unit $\alpha \rightarrow [\alpha]$. We could have provided the *Bind* and *Return* operations in the seq builder instead, but this leads to a less intuitive syntax that requires users to write let! for iteration and return for yielding.

As the Python comparison shows, the flexibility of computation expressions means that they are often close to built-in language features. The author of a concrete computation (parse, seq, async, ...) decides what syntax is appropriate. We can only provide anecdotal recommendation – for computations where the *monoidal* interface is more important, the for/yield notation fits better, while for computations where the *monadic* interface dominates we prefer let! and return.

2.3 Layered asynchronous sequences

It is often useful to combine non-standard aspects of multiple computation types. Abstractly, this has be described using monad transformers [99]. F# does not support monad transformers directly, but they provide a useful conceptual framework. For example, we migth combine non-blocking execution of asynchronous workflows with the ability to return multiple results in list comprehensions – a file download can then produce data in 1kB buffers as they become available. Such computation is captured by *asynchronous sequences* [14].

Assuming $\mathsf{Async}\,\tau$ is the type of asynchronous workflows, the composed computation can be expressed as follows (inspired by the list transformer [99]):

$$\begin{array}{lcl} \textbf{type}\ \mathsf{AsyncSeqInner}\,\tau & = & \mathsf{AsyncNil} \mid \mathsf{AsyncCons}\ \textbf{of}\ \tau \times \mathsf{Async}\,\tau \\ \textbf{type}\ \mathsf{AsyncSeq}\,\tau & = & \mathsf{Async}\,(\mathsf{AsyncSeqInner}\,\tau) \end{array}$$

When provided with a continuation, asynchronous sequence calls it with either AsyncNil (to denote the end of the sequence) or with AsyncCons that carries a value, together with the rest of the asynchronous sequence. It turns out that the flexibility of computation expression makes it possible to provide an elegant syntax for writing computations of this type:

```
let rec urlPerSecond n = asyncSeq {
   do! Async.Sleep 1000
   yield getUrl i
   yield! iterate (i + 1) }
```

```
let pagePerSecond urls = asyncSeq {
   for url in urlPerSecond 0 do
      let! html = asyncFetch url
      yield url, html }
```

The urlPerSecond function creates an asynchronous sequence that produces one URL per second. It uses bind (do!) of the asynchronous workflow monad to wait one second and then composition of asynchronous sequences, together with yield to produce the next URL. The pagePerSecond function uses for to iterate over (bind on) an asynchronous sequence and then let! to wait for (bind on) an asynchronous workflow. The for loop is asynchronous and lazy – it is run each time the caller asks for the next result.

Asynchronous sequences form a monad and so we could use the standard notation for monads with just let! and return. We would then need explicit lifting function that turns an asynchronous workflow into an asynchronous sequence that returns a single value. However, F# computation expressions allow us to do better. We can define both For and Bind with the following types:

$$\begin{array}{rcl} \mathsf{asyncSeq.For} & : & \mathsf{AsyncSeq}\,\alpha \rightarrow (\alpha \rightarrow \mathsf{AsyncSeq}\,\beta) \rightarrow \mathsf{AsyncSeq}\,\beta \\ \mathsf{asyncSeq.Bind} & : & \mathsf{Async}\,\alpha \qquad\ \rightarrow (\alpha \rightarrow \mathsf{AsyncSeq}\,\beta) \rightarrow \mathsf{AsyncSeq}\,\beta \end{array}$$

We omit the translation of the above example – it is a straightforward variation on what we have seen so far. A more important point is that we can again benefit from the fact that operations of the computation builder are not restricted to a specific type (such as *Bind* for some monad M).

As previously, the choice of the syntax is left to the author of the computation. Here, asynchronous sequences are an additive monad and so we use for/yield. Underlying asynchronous workflows are just monads, so it makes sense to add let! that automatically lifts a workflow to an asynchronous sequence.

An important aspect of realization that asynchronous sequences can be described using a monad transformer means that certain laws hold. In §4.3 we show how these map to the computation expression syntax.

2.4 Applicative formlets

Our last example uses a computation based on *applicative functors* [2], which is a weaker (and thus more common) abstraction than monads. The difference between applicative and monadic computations is that monadic computation can perform different effects depending on values obtained earlier during the computation. On the other hand, the structure of effects of applicative computation is fully determined by its structure.

In other words, it is not possible to choose which computation to run (using let! or do!) based on values obtained in previous let! bindings. The following example demonstrates this using a web form abstraction called formlets [99]:

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

The computation describes two aspects – the rendering and the processing of entered values. The rendering phase uses the fixed structure to produce HTML with text-box and drop-down elements. 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 the form needs to be known without having access to specific values. The syntax uses parallel binding (**let!**... **and**...), which binds a fixed number of independent computations. The rest of the computation cannot contain other (applicative) bindings.

There are two equivalent ways of defining applicative functors. We use the less common style which uses two operations. *Merge* of type $F\alpha \rightarrow F\beta \rightarrow F(\alpha \times \beta)$ represents composition of structure (without any knowledge of specific values) and *Map* of type $F\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow F\beta$ transforms the (pure) value. The computation expression from the previous example is translated as follows:

```
formlet.Map
  ( formlet.Merge(Formlet.textBox, Formlet.dropDown ["Male"; "Female"]),
    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 HTML. The rest of the computation is turned into a pure function passed to *Map*. Note that the translation allows uses beyond applicative functors. The let!... and... syntax can be used with monads to write zip comprehensions [99], which are useful for parsing, parallelism and more [99].

Applicative functors were first introduced to support *applicative* programming style where monads are not needed. The *idiom brackets* notation [99] fits that purpose better. We find that computation expressions provide a useful alternative for more complex code and fit better with the impure nature of F#.

3 Semantics of computation expressions

The F# language specification [17] documents computation expressions as a purely syntactic mechanism. They are desugared before type-checking, which is then performed on the translated code using standard F# typing rules. Similarly to Haskell's rebindable syntax [99], this provides more flexibility and allows the users to invent previously unforseen abstractions.

In this paper, we look at computation expressions from a different perspective. We relate them to standard abstract computation types. In §3.2, we present a new typing that captures such common uses and would make the system more robust by supporting better error messages and disallowing non-standard uses.

3.1 Syntax

The full syntax of computation expressions is given in the language specification, but the following lists all important constructs that we consider in this paper:

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

We do not include do! which can be easily expressed in terms of the let! construct. To accommodate the applicative syntax, we use a syntactic category *binds* to express one or more variable bindings.

For space reasons, we also omit imperative while and exception handling constructs. Both of these are an important part of computation expressions. The design principle is that the user should be able to wrap any valid F# code in a computation block and augment it with non-standard computational aspect, while preserving the semantics (including exception handling).

3.2 Typing

The typing rules in Figure 1 are written using three judgments. Standard F# expressions are typed using $\Gamma \vdash \mathit{expr} : \tau$. Computation expressions always return computation of type $M\tau$ and are typed using $\Gamma \Vdash_\sigma \mathit{cexpr} : M\tau$. Finally, we use a helper judgement $\Gamma \triangleright_\sigma \mathit{binds} : M\Sigma$ to check bindings of multiple computations. The judgement produces a variable context with newly bound variables, wrapped in the type M of the bound computations.

The latter two are parameterized by the type of the computation expression builder (such as seq or async). The operations supported by the builder determine which syntactic constructs are enabled. Typing rules that require a certain operation have a side-condition on the right, which specifies the requirement.

In most of the side-conditions, the functions are universally quantified over the type of values (written as α, β). This captures the fact that computation

$$\boxed{\Gamma \vdash \mathit{expr} : \tau} \quad \text{and} \quad \boxed{\Gamma \rhd_\sigma \mathit{binds} : M\Sigma}$$

$$\text{(run)}\ \frac{\Gamma \vdash \mathit{expr} : \sigma \qquad \Gamma \Vdash_\sigma \mathit{cexpr} : M\tau}{\Gamma \vdash \mathit{expr}\ \{\ \mathit{cexpr}\ \} : N\tau} \qquad \begin{array}{l}(\forall\alpha : \sigma.\mathit{Run}\ :\ D\alpha \to N\alpha\\ \ \forall\alpha : \sigma.\mathit{Delay} : (\mathsf{unit} \to M\alpha) \to D\alpha)\end{array}$$

$$\text{(bind-one)}\ \frac{\Gamma \vdash \mathit{expr} : M\tau}{\Gamma \rhd_\sigma v = \mathit{expr} : M(v\!:\!\tau)}$$

$$\text{(bind-par)}\ \frac{\Gamma \vdash \mathit{expr} : \tau \qquad \Gamma \rhd_\sigma \mathit{binds} : M\Sigma}{\Gamma \rhd_\sigma v = \mathit{expr}\ \textbf{and}\ \mathit{binds} : M(\Sigma, v\!:\!\tau)} \qquad \begin{array}{l}(\forall\alpha,\beta : \sigma.\mathit{Merge}\ :\\ \quad M\alpha \to M\beta \to M(\alpha\times\beta))\end{array}$$

$$\boxed{\Gamma \Vdash_\sigma \mathit{cexpr} : M\tau}$$

$$\text{(let)}\ \frac{\Gamma \vdash \mathit{expr} : \tau_1 \qquad \Gamma, v\!:\!\tau_1 \Vdash_\sigma \mathit{cexpr} : M\tau_2}{\Gamma \Vdash_\sigma \textbf{let}\ v = \mathit{expr}\ \textbf{in}\ \mathit{cexpr} : M\tau_2}$$

$$\text{(bind)}\ \frac{\Gamma \rhd_\sigma \mathit{binds} : M\Sigma \qquad \Gamma, \Sigma \Vdash_\sigma \mathit{cexpr} : N\tau}{\Gamma \Vdash_\sigma \textbf{let!}\ \mathit{binds}\ \textbf{in}\ \mathit{cexpr} : N\tau} \qquad \begin{array}{l}(\forall\alpha,\beta : \sigma.\mathit{Bind}\ :\\ \quad M\alpha \to (\alpha \to N\beta) \to N\beta)\end{array}$$

$$\text{(map)}\ \frac{\Gamma \rhd_\sigma \mathit{binds} : M\Sigma \qquad \Gamma, \Sigma \vdash \mathit{expr} : \tau}{\Gamma \Vdash_\sigma \textbf{let!}\ \mathit{binds}\ \textbf{in return}\ \mathit{expr} : N\tau} \qquad \begin{array}{l}(\forall\alpha,\beta : \sigma.\mathit{Map}\ :\\ \quad M\alpha \to (\alpha \to \beta) \to N\beta)\end{array}$$

$$\text{(for)}\ \frac{\Gamma \vdash \mathit{expr} : M\tau_1 \qquad \Gamma, v\!:\!\tau_1 \Vdash_\sigma \mathit{cexpr} : N\tau_2}{\Gamma \Vdash_\sigma \textbf{for}\ v\ \textbf{in}\ \mathit{expr}\ \textbf{do}\ \mathit{cexpr} : N\tau_2} \qquad \begin{array}{l}(\forall\alpha,\beta : \sigma.\mathit{For}\ :\\ \quad M\alpha \to (\alpha \to N\beta) \to N\beta)\end{array}$$

$$\text{(return-val)}\ \frac{\Gamma \vdash \mathit{expr} : \tau}{\Gamma \Vdash_\sigma \textbf{return}\ \mathit{expr} : M\tau} \qquad (\forall\alpha : \sigma.\mathit{Return}\ :\ \alpha \to M\alpha)$$

$$\text{(return-comp)}\ \frac{\Gamma \vdash \mathit{expr} : M\tau}{\Gamma \Vdash_\sigma \textbf{return!}\ \mathit{expr} : N\tau} \qquad (\forall\alpha : \sigma.\mathit{ReturnFrom}\ :\ M\alpha \to N\alpha)$$

$$\text{(seq)}\ \frac{\Gamma \Vdash_\sigma \mathit{cexpr}_1 : M\tau_1 \qquad \Gamma \Vdash_\sigma \mathit{cexpr}_2 : N\tau_2}{\Gamma \Vdash_\sigma \mathit{cexpr}_1; \mathit{cexpr}_2 : L\tau_1} \qquad \begin{array}{l}(\forall\alpha : \sigma.\mathit{Delay}\ :\ (\mathsf{unit} \to N\alpha) \to D\alpha\\ \ \forall\alpha : \sigma.\mathit{Combine} : M\tau_1 \to D\alpha \to L\alpha)\end{array}$$

$$\text{(zero)}\ \frac{\Gamma \vdash \mathit{expr} : \mathsf{unit}}{\Gamma \Vdash_\sigma \mathit{expr} : M\tau} \qquad (\sigma.\mathit{Zero}\ :\ \mathsf{unit} \to M\tau)$$

Fig. 1. Typing rules for computation expressions

should not restrict the values that users can work with. However, this is not the case in the rules (*seq*) and (*zero*). Here, we can only require that a specific instantiation is available – the reason is that these operations may be used in two different ways. As discussed in §2.1, for monads the result of *Zero* and the first argument of *Combine* are restricted to $M\,\mathsf{unit}$. They can be universally quantified only if the computation is monoidal §2.2.

Another notable aspect of the typing is that a single computation expression may use multiple computation types (written M, N, L and D). In *Bind* and *For*, the type of bound argument is M, but the resulting computation is N (we require that bind returns the same type of computation as the one produced

$$
\begin{array}{rcl}
\mathit{expr}\ \{\ \mathit{cexpr}\ \} &=& \textbf{let}\ m = \mathit{expr}\ \textbf{in}\ m.\mathsf{Run}(m.\mathsf{Delay}(\textbf{fun}\ () \rightarrow [\![\ \mathit{cexpr}\ ]\!]_m))\\
[\![\ \textbf{let}\ v = \mathit{expr}\ \textbf{in}\ \mathit{cexpr}\ ]\!]_m &=& \textbf{let}\ v = \mathit{expr}\ \textbf{in}\ [\![\ \mathit{cexpr}\ ]\!]_m\\
[\![\ \textbf{let!}\ \mathit{binds}\ \textbf{in}\ \mathit{cexpr}\ ]\!]_m &=& m.\mathsf{Bind}(\langle\!\langle \mathit{binds} \rangle\!\rangle_m, \textbf{fun}\ \langle \mathit{binds} \rangle \rightarrow [\![\ \mathit{cexpr}\ ]\!]_m)\\
[\![\ \textbf{let!}\ \mathit{binds}\ \textbf{in}\ \textbf{return} \mathit{expr}\ ]\!]_m &=& m.\mathsf{Map}(\langle\!\langle \mathit{binds} \rangle\!\rangle_m, \textbf{fun}\ \langle \mathit{binds} \rangle \rightarrow \mathit{expr})\\
[\![\ \textbf{for}\ v\ \textbf{in}\ \mathit{expr}\ \textbf{do}\ \mathit{cexpr}\ ]\!]_m &=& m.\mathsf{For}(\mathit{expr}, \textbf{fun}\ () \rightarrow [\![\ \mathit{cexpr}\ ]\!]_m)\\
[\![\ \textbf{return}\ \mathit{expr}\ ]\!]_m &=& m.\mathsf{Return}(\mathit{expr})\\
[\![\ \textbf{return!}\ \mathit{expr}\ ]\!]_m &=& m.\mathsf{ReturnFrom}(\mathit{expr})\\
[\![\ \mathit{cexpr}_1; \mathit{cexpr}_2\ ]\!]_m &=& m.\mathsf{Combine}([\![\ \mathit{cexpr}_1\ ]\!]_m,\ m.\mathsf{Delay}(\textbf{fun}\ () \rightarrow [\![\ \mathit{cexpr}_2\ ]\!]_m))\\
[\![\ \mathit{expr}\ ]\!]_m &=& \mathit{expr};\ m.\mathsf{Zero}()\\
\langle\!\langle v = \mathit{expr} \rangle\!\rangle_m &=& \mathit{expr}\\
\langle\!\langle v = \mathit{expr}\ \textbf{and}\ \mathit{binds} \rangle\!\rangle_m &=& m.\mathsf{Merge}(\mathit{expr}, [\![\ \mathit{binds}\ ]\!]_m)\\
\langle v = \mathit{expr} \rangle &=& v\\
\langle v = \mathit{expr}\ \textbf{and}\ \mathit{binds} \rangle &=& v, \langle \mathit{binds} \rangle
\end{array}
$$

Fig. 2. Translation rules for computation expressions

by the function). This corresponds to the typing used by computations arising from monad transformers §2.3. Although combining multiple computation types is not as frequent, computations often have a delayed version which we write as D. This is an important consideration for impure langauges such as F# §3.4.

Finally, we omitted typing for yield and yield! because it is similar to the typing of return and return! (using *Yield* and *YieldFrom* operations, respectively).

3.3 Translation

The translation is defined as a mapping $[\![-]\!]_m$ that is parameterized by a variable m which refers to the current instance of a computation builder. This parameter is later used in the translation to invoke members of the builder, such as $m.\mathsf{Return}(\ldots)$. Multiple variable bindings are translated using $\langle\!\langle \mathit{binds} \rangle\!\rangle_m$ and we define a helper mapping $\langle \mathit{binds} \rangle$ that turns bindings into a simple pattern that can be used to decompose a tuple constructed by merging computations using the *Merge* operation.

According to the F# specification, a particular construct of computation expression syntax is allowed only when the static type of the computation builder defines members that are required by the translation. It is easy to check that our typing rules guarantee that a well-typed computation expression can always be translated to a well-typed F# expression.

Careful readers have already noticed that our definition of $[\![-]\!]_m$ is ambiguous. The let! binding followed by return can be translated in two different ways. In the real implementation, the translation using *Map* is preferred, but we do not specify this in the paper. The reason is that the laws in §4.2 require the two translations to be equivalent. For monads, this equivalence is easy to see by considering the definition of *Map* in terms of *Bind* and *Return*.

In earlier discussion, we omitted the *Run* and *Delay* members in the translation of $\mathit{expr}\ \{\ \mathit{cexpr}\ \}$. The next section discusses these two in more details.

3.4 Delayed computations

We already mentioned that side-effects are an important consideration when adding sequencing to monadic comptuations §2.1. In effectful languages, we need to distinguish between two types of monads. We use the term *monadic computation* for monads that represent a delayed computation such as asynchronous workflows or lazy list comprehensions; the term *monadic containers* will be used for monads that represent a wrapped non-delayed value (such as the option type, non-lazy list or the identity monad).

Effects and monadic computations. The defining feature of *monadic computations* is that they permit a *Delay* operation of type (unit $\rightarrow M\alpha) \rightarrow M\alpha$ that does not perform the effects associated with the function used as an argument. For example, in the continuation monad (underlying asynchronous workflows), the operation builds a computation that takes a continuation – and so the effects are only run when the continuation is provided.

Before going further, we revist the translation of asynchronous workflows using the full set of rules to show how *Run* and *Delay* are used. Consider the the following simple computation with a corresponding translation:

```
let answer = async {                 let answer = async.Run(async.Delay(fun () →
  printfn "Welcome..."                 printfn "Welcome..."
  return 42 }                          async.Return(42) ))
```

For monadic computations such as asynchronous workflows, we do not expect that the defining answer will print "Welcome". This is achieved by the wrapping specified in the translation rule for the *expr* { *cexpr* } expression. As already mentioned, the result of *Delay* is a

In this case, the result of *Delay* is a computation A int that encapsulates the delayed effect. For monadic containers, the *Run* function is a simple identity – contrary to what the name suggests, it does not run the computation (although that might be an interesting use beyond standard abstract computations). The need for *Run* becomes obvious when we look at monadic containers.

Effects and monadic containers. For monadic containers, it is impossible to define a *Delay* operation that does not perform the effects and has a type (unit $\rightarrow M\alpha) \rightarrow M\alpha$, because the resulting type has no way of capturing unevaluated code. However, the (*seq*) typing rule in Figure 1 permits an alternative typing. Consider the following example using the Maybe (option) monad:

```
maybe { if b = 0 then return! None
        printfn "Calculating..."
        return a / b }
```

Using the same translation rules, *Run*, *Delay* and *Delay* are inserted as follows:

```
maybe.Run(maybe.Delay(fun () → maybe.Combine
   ( (if b = 0 then maybe.ReturnFrom(None) else maybe.Zero()),
     maybe.Delay(fun () → printfn "Calculating..."
                           maybe.Return(a / b)) ) ))
```

The key idea is that we do not have to use the type $M\alpha$ for representing delayed computations, but can instead use two different types throughout the code. $M\alpha$ for values representing evaluated containers and $\mathsf{unit} \to M\alpha$ for delayed computations. The operations have the following types:

$$\begin{array}{lcl} \mathit{Delay} & : & (\mathsf{unit} \to M\alpha) \to (\mathsf{unit} \to M\alpha) \\ \mathit{Run} & : & (\mathsf{unit} \to M\alpha) \to M\alpha \\ \mathit{Combine} & : & M\,\mathsf{unit} \to (\mathsf{unit} \to M\alpha) \to M\alpha \end{array}$$

Here, the *Delay* operation becomes just an identity that returns the function created by the translation. In the translation, the result of *Delay* can be passed either to *Run* or as the second argument of *Delay*, so these need to be changed accordingly. The *Run* function now becomes important as it turns the delayed function into a value of the expected type $M\alpha$.

Unified treatment of effects. In the typing rules §3.2, we did not explicitly list the two options, because they can be generalized. We require that the result of *Delay* is some (possibly different) abstract type $D\alpha$ representing delayed computations. For monadic computations, the type is just $M\alpha$ and for monadic containers, it is $\mathsf{unit} \to M\alpha$. Our typing is even more flexible, as it allows usage of multiple different computation types – but treatment of effects is one example where this additional flexibility is necessary.

Finally, it should be noted that we used a slight simplification. The actual F# implementation does not strictly require *Run* and *Delay* in the translation of *expr* { *cexpr* }. They are only used if they are present.

4 Computation expression laws

Although computation expressions are not tied to any specific abstract computation type, we showed that they are usually used with well-known abstractions like monads, monad transofrmers or applicative functors.

This means three good things. First, we get better understanding of what computations can be encoded (and how). Second, we can add a more precise typing §3.2. Third, we know that certain syntactic transformations (refactorings) preserve the meaning of computation. This section looks at the last point.

This section assumes that there are no side-effects and we ignore *Run* and *Delay*. These can be added to the picture, but it complicates the presentation.

4.1 Monoid and semi-group laws

We start by looking at the simplest possible structure. A semigroup $(S, \circ)$ consists of a set S and an associative binary operation $\circ$ meaning that $a \circ (b \circ c) = (a \circ b) \circ c$. A computation expression corresponding to a semigroup defines only *Combine* (of a type $M\alpha \to M\alpha \to M\alpha$). To allow appropriate syntax, we also add *YieldFrom* which is just the identity function. The associativity implies the following syntactic equivalence:

$$\mathsf{m}\ \{\ \mathit{cexpr}_1;\ \mathit{cexpr}_2;\ \mathit{cexpr}_3\ \}\ \equiv\ \mathsf{m}\ \{\ \textbf{yield!}\ \mathsf{m}\ \{\mathit{cexpr}_1;\ \mathit{cexpr}_2;\};\ \mathit{cexpr}_3\ \}$$

For semigroups, the syntax is rather limited, but given a value x of type $M\tau$ it is possible to write $\mathsf{yield!}\ x$ to return the value.

A monoid $(S, \circ, \epsilon)$ is a semigroup $(S, \circ)$ with an identity element ϵ meaning that for all values $a \in S$ it holds that $\epsilon \circ a = a = a \circ \epsilon$. The identity element can be added to computation builder as the *Zero* member. This operation is used when a computation uses conditional without else branch. Thus we get:

$$\begin{array}{l}\mathsf{m}\ \{\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ \mathit{cexpr}_1 \\ \quad \mathit{cexpr}_2\ \}\end{array} \equiv \mathsf{m}\ \{\ \mathit{cexpr}_2\ \} \equiv \begin{array}{l}\mathsf{m}\ \{\ \mathit{cexpr}_2 \\ \quad \textbf{if}\ \mathsf{false}\ \textbf{then}\ \mathit{cexpr}_1\ \}\end{array}$$

Although these are simple laws, they can be used to reason about list comprehensions. The associativity means that we can move a part of computation expression (that uses yield! repeatedly) into a separate computation. To use the identity law, consider a recursive function that generates numbers up to 100:

$$\begin{array}{l}\textbf{let rec}\ \mathsf{range}\ n = \\ \quad \mathsf{seq}\ \{\ \textbf{yield}\ n \\ \quad\quad\quad \textbf{if}\ n < 100\ \textbf{then}\ \textbf{yield!}\ \mathsf{range}\ (n+1)\ \}\end{array}$$

Here, we can see that when $n = 100$, the body is equivalent to just $\mathsf{m}\ \{\ \mathsf{yield}\ 100\ \}$. Indeed, this is an expected property of the if construct – the law guarantees that the property holds even for an if construct that is reinterpreted by some (monoidal) computation expression.

4.2 Monad and additive monad laws

Monad laws are well-understood and the corresponding equivalent computation expressions do not significantly differ from the laws about Haskell's do notation:

$$\begin{array}{rcl}\mathsf{m}\ \{\ \textbf{let!}\ y = \mathsf{m}\ \{\ \textbf{return}\ x\ \}\ \textbf{in}\ \mathit{cexpr}\ \} & \equiv & \mathsf{m}\ \{\ \textbf{let}\ y = x\ \textbf{in}\ \mathit{cexpr}\ \} \\ \mathsf{m}\ \{\ \textbf{let!}\ x = c\ \textbf{in}\ \textbf{return}\ x\} & \equiv & \mathsf{m}\ \{\ \textbf{return!}\ c\}\end{array}$$

$$\begin{array}{l}\mathsf{m}\ \{\ \textbf{let!}\ x = \mathsf{m}\ \{\ \textbf{let!}\ y = c\ \textbf{in}\ \mathit{cexpr}_1\ \}\ \textbf{in}\ \mathit{cexpr}_2\ \} \equiv \\ \equiv \mathsf{m}\ \{\ \textbf{let!}\ y = c\ \textbf{in}\ \textbf{let!}\ x = \mathsf{m}\ \{\ \mathit{cexpr}_1\ \}\ \textbf{in}\ \mathit{cexpr}_2\ \}\end{array}$$

However, there is more to be said about the *Map* operation in the translation and about laws of additive monads (MonadPlus typeclass in Haskell).

Alternative translations. When discussing the translation rules §3.3, we noted that the rules are ambiguous when both *Map* and *Bind* operations are present. The following can be translated both monadically and applicatively:

$$\mathsf{m}\ \{\ \textbf{let!}\ x = c\ \textbf{in}\ \textbf{return}\ \mathit{expr}\ \}$$

The two translations are shown below. Assuming that our computation is a monad, this is a well-known definition of *Map* in terms of *Bind* and *Return*:

$$\mathsf{m.Map}(x, \textbf{fun}\ x \to \mathit{expr}) \equiv \mathsf{m.Bind}(x, \textbf{fun}\ x \to \mathsf{m.Return}(\mathit{expr}))$$

More generally, if a computation builder defines both *Map* and *Bind* (even if they are not based on a monad), we require this equation to guarantee that the two possible translations produce equivalent computations.

Additive monads. Additive monads are computations that combine monad with the monoidal structure. As shown earlier §2.2, these can be embedded using let!/return or using for/yield (depending on which aspect is "more important").

The set of laws required for such computations is not fully resolved [99]. A more generally accepted law is *left distributivity* – applying monoidal operation and then binding is equivalent to binding on two computations and then combining the results. In terms of computation builder operations:

$$\mathsf{m.For}(\mathsf{m.Combine}(a, b), f) \;\equiv\; \mathsf{m.Combine}(\mathsf{m.For}(a, f), \mathsf{m.For}(b, f))$$

We intentionally use the *For* operation (corresponding to the for keyword), because this leads to the following intuitive syntactic equality:

$$\begin{array}{ll} \mathsf{m}\;\{\;\textbf{for}\;x\;\textbf{in}\;\mathsf{m}\;\{\;\mathit{cexpr}_1;\;\mathit{cexpr}_2\;\}\;\textbf{do} & \equiv\;\mathsf{m}\;\{\;\textbf{for}\;x\;\textbf{in}\;\mathsf{m}\;\{\;\mathit{cexpr}_1\;\}\;\textbf{do}\;\mathit{cexpr} \\ \quad\quad \mathit{cexpr}\;\} & \quad\quad\;\;\textbf{for}\;x\;\textbf{in}\;\mathsf{m}\;\{\;\mathit{cexpr}_2\;\}\;\textbf{do}\;\mathit{cexpr}\;\} \end{array}$$

If we read the code as an imperative looping construct (without the computational reinterpretation), then this is, indeed, a valid law about for loops.

Another law that is sometimes required about additive monads is *left catch*. It states that combining a computation that immediately returns a value with any other computation results in a computation that just returns the value:

$$\mathsf{m.Combine}(\mathsf{m.Return}(v), a) \quad\equiv\quad \mathsf{m.Return}(v)$$

This time, we intentionally used the *Return* member instead of *Yield*, because the law corresponds to the following syntactic equivalence:

$$\mathsf{m}\;\{\;\textbf{return}\;v;\;\mathit{cexpr}\;\} \quad\equiv\quad \mathsf{m}\;\{\;\textbf{return}\;v\;\}$$

The fact that *left distributivity* corresponds to an intuitive syntactic equality about for/yield while *left catch* corresponds to a syntactic equality about let!/return provides a useful guidance for choosing between the two syntactic options. The former is appropriate for list comprehensions (and other collections), while the latter is appropriate for example for the left-biased option (Maybe) monad, imperative computations [99] or software transactional memory.

4.3 Monad transformers

There are multiple ways of composing or layering monads [99, 98]. Monad transformers are perhaps the most widely known technique. A monad transformer is a type constructor $T\,m$ together with a *Lift* operation. For some monad M the operation has a type $M\,\alpha \to T\,M\,\alpha$. and it turns a computation in the underlying monad into a computation in the composed monad.

The result of monad transformer is also a monad. This means that we can use the usual syntactic sugar for monads, such as the do notation in Haskell. However, a more specific notation can use the additional *Lift* operation.

We demonstrated encoding of syntax for composed monads when discussing asynchronous sequences §2.3. An asynchronous sequence $\mathsf{AsyncSeq}\,\alpha$ is a computation obtained by applying the list monad transformer [97] to the asynchronous workflow $\mathsf{Async}\,\alpha$ monad. Asynchronous sequences are *additive monads* satisfying the left distributivity law, so we choose the for/yield syntax for working with the composed computation. We also provided an additional *Bind* operation to support awaiting a single asynchronous workflow using the let! construct. This operation is defined in terms of *Lift* of the monad transformer and *For* (monadic bind) of the composed computation:

$$\mathsf{asyncSeq.Bind}(a, f) = \mathsf{asyncSeq.For}(\mathsf{asyncSeq.Lift}(a), f)$$

There are two laws that hold about monad transforers. To avoid confusion, we use asynchronous workflows and sequences in the explanation, but we could easily generalize. The first law states that composing *Return* of asynchronous workflows with *Lift* should be equivalent to the *Yield* of asynchronous sequences. The other states that *Lift* distributes over monadic bind.

Our syntax always combines *Lift* with *For* and so there are multiple syntactic equivalences that follow from the laws. The following are the most direct (the first one also relies on right identity for monads):

$$\mathsf{asyncSeq}\ \{\ \textbf{let!}\ x = \mathsf{async}\ \{\ \textbf{return}\ v\ \}\ \textbf{in return}\ x\ \}\ \equiv\ \mathsf{asyncSeq}\ \{\ \textbf{return}\ v\ \}$$

$$\begin{aligned}&\mathsf{asyncSeq}\ \{\ \textbf{let!}\ x = \mathsf{async}\ \{\ \textbf{let!}\ y = c\ \textbf{in}\ cexpr_1\ \}\ \textbf{in}\ cexpr_2\ \}\ \equiv\\ \equiv\ &\mathsf{asyncSeq}\ \{\ \textbf{let!}\ y = c\ \textbf{in let!}\ x = \mathsf{async}\ \{\ cexpr_1\ \}\ \textbf{in}\ cexpr_2\ \}\end{aligned}$$

The first equation returns the value v without any asynchronous waiting in both cases (although, in presence of side-effects, this is made more complicated by cancellation). The second equation is more subtle. The left-hand side awaits a single asynchronous workflow that first awaits c and then does more work. The right-hand side awaits c lifted to an asynchronous sequence and then awaits the rest (again, lifted into an asynchronous sequence).

4.4 Applicative computations

The last type of computations that we discussed §2.4 is *applicative functor*. We use the less common definition (called Monoidal by McBride and Paterson [99]). The definition consists of *Map* and *Merge* operations, together with a unit computation. We use the unit to define *Zero* – in the translation it will only be used in computations that contain some unit-returning expression, such as ().

The identity law guarantees that merging with a unit and then projecting the non-unit value produces an equivalent computation:

$$\begin{array}{l}\mathsf{f}\ \{\ \textbf{let!}\ x = \mathsf{f}\ \{\ ()\ \}\\ \quad\ \textbf{and}\ y = c\ \textbf{in return}\ y\ \}\}\end{array}\ \equiv\ c\ \equiv\ \begin{array}{l}\mathsf{f}\ \{\ \textbf{let!}\ x = c\\ \quad\ \textbf{and}\ y = \mathsf{f}\ \{\ ()\ \}\ \textbf{in return}\ x\ \}\}\end{array}$$

The naturality law specifies that *Merge* distributes over *Map*, which translates to the following equivalence (assuming x_1 not free in $expr_2$ and vice versa):

$$\begin{array}{l}\mathsf{f}\ \{\ \textbf{let!}\ y_1 = \mathsf{f}\ \{\ \textbf{let!}\ x_1 = c_1\ \textbf{in return}\ expr_1\ \}\\ \quad\ \textbf{and}\ y_2 = \mathsf{f}\ \{\ \textbf{let!}\ x_2 = c_2\ \textbf{in return}\ expr_2\ \}\ \textbf{in}\ expr\ \}\end{array}\ \equiv$$

$$\equiv\ \mathsf{f}\ \{\ \textbf{let!}\ x_1 = c_1\ \textbf{and}\ x_2 = c_2\ \textbf{in let}\ y_1, y_2 = expr_1, expr_2\ \textbf{in}\ expr\ \}$$

As with the earlier syntactic rules, we can leave out the non-standard aspect of the computations and read them as ordinary functional code and get correct and expected laws. This means that the laws, again, guarantee that intuition about the syntax used by computation expressions will be correct.

Finally, the *Merge* operation is also required to be associative – this does not have any corresponding syntax, but it means that the user does not need to know implementation details of the compiler – it does not matter whether let!... and... is left-associative or right-associative.

5 Conclusions

Related work - do notation works for any monad - which makes it more reusable but weaker

Acknowledgements. We are grateful to Dominic Orchard, Alan Mycroft, Sam Lindley, anonymous reviewers of previous draft and the audience of TFP 2012.