The F# Computation Expression 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. Program logic can often be structured using abstract computation types such as monoids, monad transformers or applicative functors. Functional programmers use those abstractions directly while mainstream 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, hardwired language features, and an inconvenient but flexible libraries?

F# *computation expressions* answer this question in the affirmative. Unlike the "do" notation in Haskell, computation expressions are not tied to a single kind of abstraction. They support a wide range of abstractions, depending on what operations are available. F# also provides greater syntactic flexibility leading to a more intuitive syntax, without resorting to full macro-based meta-programming.

We present computation expressions in a principled way, developing a type system that captures the semantics of the calculus. We demonstrate how computation expressions structure well-known abstractions including monoidal list comprehensions, monadic parsers, applicative formlets and asynchronous sequences based on monad transformers.

1 Introduction

Computations with non-standard aspects like non-determinism, effects, asynchronicity or their combinations can be captured using a variety of abstract computation types. In Haskell, we write such computations using a mix of combinators and syntactic extensions like monad comprehensions [5] and "do" notation. Languages such as Python and C# emphasize the syntax and provide single-purpose support e.g. for asynchrony [1] and list generators.

Using such abstractions can be made simpler and more intuitive if we employ a general syntactic machinery. F# computation expressions provide *uniform* syntax that supports monoids, monads [22], monad transformers [10] and applicative functors [13]. They reuse familiar syntax including loops and exception handling – the laws of underlying abstractions guarantee that these constructs preserve intuition about code. At the same time, the mechanism is *adaptable* and enables appropriate syntax depending on the abstraction.

Most languages, including Haskell, Scala, C#, JavaScript and Python have multiple syntactic extensions that improve computational expressivity: queries, iterators, comprehensions, asynchronous computations are just a few. However,

"syntactic budget" for such extensions is limited. Haskell already uses three notations for comprehensions, monads and arrows [15]. C# and Scala have multiple notations for queries, comprehensions, asynchronicity and iterators. The more we get with one mechanism, the better. As we show, computation expressions give a lot for relatively low cost – notably, without resorting to full-blown macros.

Some of the technical aspects of the feature have been described before[3] [20], but this paper is novel in that it uses more principled approach by developing a new type system and relating the mechanism to well-known abstractions.

Practical examples. We demonstrate the breadth 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 that important syntactic equalities hold as a result (§4).

Syntax and typing. We give typing rules that capture idiomatic uses of computation expressions (§3.2), extend the translation to support applicative functors (§2.4) and discuss the treatment of effects (§3.4) needed in impure languages.

We believe that software artifacts in programming language research matter [9], so all code can be run at: `http://tryjoinads.org/computations`. The syntax for applicative functors is a reserch extension; other examples require F# 2.0.

2 Computation expressions by example

Computation expressions are blocks of code that represent computations with a non-standard aspect such as laziness, asynchronicity, state or other. The code inside the block is re-interpreted using a *computation builder*, which is a record of operations that define the semantics, but also syntax available in the block.

Computation expressions mirror the standard F# syntax (let binding, loops, exception handling), but support additonal computational constructs. For example `let!` represents the computational (monadic) alternative of let binding.

We first introduce the syntax and mapping to the underlying operations informally, but both are made precise later (§3). Readers unfamiliar with F# may find additional explanation in previous publications [20]. To show the breadth of applications, we look at five examples arising from different abstractions.

2.1 Monadic asynchronous workflows

Asynchronous workflows [19] allow writing non-blocking I/O using a mechanism based on the *continuation monad* (with error handling etc.) The following example compares F# code with an equivalent in C# using a single-purpose feature:

[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.

```
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;
}
```

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 {...} block, where async is an identifier that refers to a library-defined computation builder.

The computation builder async is an F# object with instance members such as *async.Bind*. The members determine which of the pre-defined keywords are allowed – e.g. *Bind* member enables let! which represents (monadic) binding. *Bind* also enables the do! e expression, which is a shortcut for let! () = e. Finally, the return keyword is mapped to the *Return* operation:

```
async.Bind(fetchAsync(url), fun html →
    async.Bind(Async.Sleep 1000, fun () →
        async.Return(html.Length)))
```

The *Bind* and *Return* operations form a monad. As usual, *Return* has a type $\alpha \to A\alpha$ and the required type of *Bind* is $A\alpha \times (\alpha \to A\beta) \to A\beta$ (we write α, β for universally qualified type variables and τ as for concrete types) [4].

Sequencing and effects. Effectful expressions in F# return a value () which is the only value of type unit. Assuming e_1 has a type unit, we can sequence expression using $e_1; e_2$. We can also write effectful if condition without the else clause (which implicitly returns the unit value () in the false case). Both have an equivalent 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 the page and returning it. The translation uses additional operations – *Zero* represents monadic unit value, *Combine* corresponds to the ";" operator and *Delay* embeds an effectful expression in a (delayed) computation. For monads, these can be defined in terms of *Bind* and *Return*, but this is not the case for all computations (e.g. monoidal computations discussed in §2.2 require different definitions).

We also use the return! keyword, which returns the result of a computation and requires an operation *ReturnFrom* of type $A\alpha \to A\alpha$. 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.

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

```
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)))))
```

Zero has a type unit $\rightarrow A$ unit and is inserted when a computation does not return a value, here in both branches of if. A computation returning unit can be composed with another using *Combine* which has a type $A\,\mathsf{unit} \times A\alpha \rightarrow A\alpha$ and corresponds to ";". It runs the left-hand side before returning the result of the right-hand side. Finally, *Delay*, of type $(\mathsf{unit} \rightarrow A\tau) \rightarrow A\tau$, is used to wrap any effectful computations (like printing) in the monad to avoid performing the effects before the first part of sequential computation is run.

2.2 Additive parsers and list comprehensions

Parsers or list comprehensions differ in that they may return multiple values. Such computations can be structured using additive monads (MonadPlus in Haskell). These abstractions can be used with F# computation expressions too. Interestingly, they require different typing of *Zero* and *Combine*.

Monadic parsers. For parsers, we use the same notation as previously. 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:

```
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( [] ) )))
```

Here, *Combine* represents the monoidal operation on parsers (either left-biassed or non-deterministic choice) and has the type $P\alpha \times P\alpha \rightarrow P\alpha$. Accordingly, the *Zero* operations is the unit of the monoid. It has a type unit $\rightarrow P\alpha$, representing a parser that returns no α values (rather than returning a single unit value).

For effectful sequencing of monads, it only makes sense to use unit-returning computations in the left-hand side of *Combine* and as the result of *Zero*. However, if we have a monoidal computation, we can define *Combine* that combines multiple produced values. This shows that the computation expression mechanism needs certain flexibility – the translation is the same, but the typing differs.

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

```
seq { for n in list do          for n in list :
        yield n                   yield n
        yield n * 10 }            yield n * 10
```

The computations iterate over a source list and produce two results for each input. Monad comprehensions [5] allow us to write [$n * 10$ | $n \leftarrow list$] to multiply all elements by 10, but they are not expressive enough to capture the duplication. Doing that requires rewriting the code using combinators.

The F# syntax works similarly to what we have seen for monads. The for and yield constructs are translated to *For* and *Yield* operations which have the same types as emphBind and *Return*, but provide backing for a different syntax (each keyword is mapped to a specific named operation of the builder e.g. for uses *seq.For*, so the members defined by *seq* determine which keywords are enabled):

```
seq.For(list, fun () →
  seq.Combine(seq.Yield(n), seq.Delay(fun () → seq.Yield(n * 10))) )
```

Combine concatenates multiple results and has the standard monoidal type $[\alpha] \times [\alpha] \to [\alpha]$. *For* has the type of monadic bind $[\alpha] \to (\alpha \to [\beta]) \to [\beta]$ and *Yield* has a type of monadic unit $\alpha \to [\alpha]$. 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 a built-in syntax. The author of a concrete computation (parse, seq, async, ...) chooses the appropriate syntax. For additive monads, the choice can be made based on the laws that hold §4.2.

2.3 Layered asynchronous sequences

It is often useful to combine non-standard aspects of multiple computations. This is captured by monad transformers [10]. Although F# does not support higher-kinded types, monad transformers still provide a useful conceptual framework.

For example, *asynchronous sequences* [16] combine non-blocking asynchronous execution with the ability to return multiple results – a file download can then produce data in 1kB buffers as they become available. Using Async τ as the base type, we can follow the list monad transformer [7] and define the type as:

```
type AsyncSeqInner τ = AsyncNil | AsyncCons of τ × Async τ
type AsyncSeq τ      = Async (AsyncSeqInner τ)
```

When given a continuation, an asynchronous sequence calls it with either the end of the sequence AsyncNil or with AsyncCons that carries a value together with the tail of the asynchronous sequence. The flexibility of computation expression makes it possible to provide an elegant syntax for writing such computations:

```
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's body 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:

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

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 use the fact that operations of the computation builder are not restricted to a specific type (the above *Bind* is not an ordinary binding making let! behave differently).

As previously, the choice of the syntax is left to the author of the computation. 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 the fact that asynchronous sequences can be described using a monad transformer is that certain laws hold. We discuss how these map to the computation expression syntax later (§4.3).

2.4 Applicative formlets

Applicative functors [13,11] are weaker (and thus more common) abstraction than monads. The difference between applicative and monadic computations is that a monadic computation can perform different effects depending on values obtained earlier during the computation. Conversely, the effects of an applicative computation are 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 [2]:

```
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 definitions of applicative functors. We need two operations known from the less common definition. *Merge* of type $F\alpha \times F\beta \to F(\alpha \times \beta)$ represents composition of the structure (without considering specific values) and *Map* of type $F\alpha \times (\alpha \to \beta) \to F\beta$ 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 computations composed using parallel binding are combined using *Merge*. In formlets, this determines the structure used for HTML rendering. 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 also be used with monads to write zip comprehensions [5].

Applicative functors were first introduced to support *applicative* programming style where monads are not needed. The *idiom brackets* notation [13] 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 [20] 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, but to a greater level, this provides flexibility that allows the users to invent previously unforseen abstractions.

The purely syntactic approach allows more experimentation, but does not disallow erroneous uses. In this section, we present new typing rules that capture such common uses and make the system more robust. Aside from guaranteeing idiomatic use of computation expressions, it also enables better error messages.

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:

$$
\begin{array}{lll}
\mathit{expr} & = \ldots \mid \mathit{expr}\ \{\ \mathit{cexpr}\ \} & \text{(computation expression)} \\
\mathit{binds} & = v = \mathit{expr} & \text{(single binding)} \\
 & \mid\ v = \mathit{expr}\ \textbf{and}\ \mathit{binds} & \text{(parallel binding)}
\end{array}
$$

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

We omit do! which is easily expressed using let! To accommodate the applicative syntax, *binds* is used to express one or more parallel variable bindings.

For space reasons, we also omit imperative while and exception handling constructs, but both of these are an important part of computation expressions. They allow taking existing code and wrapping it in a computation block to augment it with non-standard computational aspect.

3.2 Typing

The Figure 1 uses 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 $\Gamma \Vdash_\sigma \mathit{cexpr} : M\tau$. A helper judgement $\Gamma \rhd_\sigma \mathit{binds} : M\Sigma$ checks bindings of multiple computations and 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 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 earlier (§2.1), for monads the result of *Zero* and the first argument of *Combine* are restricted to M 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 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 (§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).

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 1. Typing rules for computation expressions

3.3 Translation

The translation is defined as a relation $[\![-]\!]_m$ that is parameterized by a variable m which refers to the current instance of a computation builder. This parameter is used to invoke members of the builder, such as $m.\mathit{Return}(\ldots)$. 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 pattern that can be used to decompose a tuple constructed by merging computations using the *Merge* operation.

As can be easily checked, our typing guarantees that a well-typed computation expression is always translated to a well-typed F# expression. The side-conditions ensure that all operations are available and have an appropriate type.

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

Fig. 2. Translation rules for computation expressions

Some readers have already noticed that our definition of $[\![-]\!]_m$ is ambiguous. The let! binding followed by return can be translated in two different ways. We intentionally do not specify the behaviour in this paper – the laws (§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 *expr* { *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, it becomes apparent that 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 lists; 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).

Monadic computations. The defining feature of *monadic computations* is that they permit a *Delay* operation of type $(\mathsf{unit} \rightarrow M\alpha) \rightarrow M\alpha$ that does not perform the effects associated with the function argument. For example, in asynchronous workflows, the operation builds a computation that waits for 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 defining answer will print "Welcome". This is achieved by the wrapping the specified computation in the translation rule for the *expr* { *cexpr* } expression.

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

Monadic containers. For monadic containers, it is impossible to define a *Delay* operation that does not perform the (untracked) side-effects and has a type $(\mathsf{unit} \to M\alpha) \to M\alpha$, 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 can use two different types – $M\alpha$ for varlues representing (evaluated) monadic containers and $\mathsf{unit} \to M\alpha$ for delayed computations. The operations then 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 $M\alpha$ (by applying it).

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 use 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. 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.

To keep the presentation in this section focused, we assume that there are no untracked side-effects (such as I/O) and we ignore *Run* and *Delay*.

4.1 Monoid and semigroup laws

We start from the simplest structures. A semigroup $(S, \circ)$ consists of a set S and a binary operation $\circ$ such that $a \circ (b \circ c) = (a \circ b) \circ c$. A computation expression corresponding to a semigroup defines only *Combine* (of type $M\alpha \times M\alpha \to M\alpha$). To allow appropriate syntax, we also add *YieldFrom* which is just the identity function (with a type $M\alpha \to M\alpha$). The associativity implies the following syntactic equivalence (we use m as a placeholder for concrete computation builder):

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

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:

$$\mathsf{m}\ \{\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ \mathit{cexpr}_1\ \mathit{cexpr}_2\ \}\ \equiv\ \mathsf{m}\ \{\ \mathit{cexpr}_2\ \}\ \equiv\ \mathsf{m}\ \{\ \mathit{cexpr}_2\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ \mathit{cexpr}_1\ \}$$

Although these are simple laws, they can be used to reason about list comprehensions. The associativity means that we can move a sub-expression 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:

```
let rec range n = seq {
  yield n
  if n < 100 then yield! range (n + 1) }
```

The law guarantees that for $n = 100$, the body equals seq { yield 100 }. This is an expected property of the if construct – the law guarantees that it holds even for if 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:

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

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

Resolving ambiguity. 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 using the *Map* operation:

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

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*:

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

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. The choice can be made based on the laws that hold.

The laws required for additive monads is not fully resolved [8]. A frequently advocated law is *left distributivity* – binding on the result of a monoidal operation is equivalent to binding on two computations and then combining the results:

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

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

$$\begin{aligned} &\mathsf{m}\ \{\ \textbf{for}\ x\ \textbf{in}\ \mathsf{m}\ \{\ cexpr_1;\ cexpr_2\ \}\ \textbf{do}\ \equiv\ \mathsf{m}\ \{\ \textbf{for}\ x\ \textbf{in}\ \mathsf{m}\ \{\ cexpr_1\ \}\ \textbf{do}\ cexpr \\ &\quad cexpr\ \} \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{for}\ x\ \textbf{in}\ \mathsf{m}\ \{\ cexpr_2\ \}\ \textbf{do}\ cexpr\ \} \end{aligned}$$

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:

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

The fact that *left catch* corresponds to an intuitive syntactic equality about let!/return while *left distributivity* corresponds to an intuitive syntactic equality about for/yield determines the appropriate syntax. The former can be used for list comprehensions (and other collections), while the latter is suitable e.g. for the option monad or the software transactional memory monad [6].

4.3 Monad transformers

There are multiple ways of composing or layering monads [10,12]. 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 transformed 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 looked at computation expression for a composed monad when discussing asynchronous sequences (§2.3). An asynchronous sequence $\mathsf{AsyncSeq}\,\alpha$ is a computation obtained by applying the list monad transformer [7] to the monad $\mathsf{Async}\,\alpha$. 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 additional *Bind* to support awaiting a single asynchronous workflow using let! 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 simplify the presentation, we use asynchronous workflows and sequences rather than showing the generalised version. 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*, so the following syntactic equivalences also require right identity for monads and function extensionality:

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

$$\begin{aligned}\mathsf{asyncSeq}\ \{\ \textbf{let!}\ x = \mathsf{async}\ \{\ \textbf{let!}\ y = c\ \textbf{in}\ \mathit{cexpr}_1\ \}\ \textbf{in}\ \mathit{cexpr}_2\ \}\ &\equiv\\ \equiv\ \mathsf{asyncSeq}\ \{\ \textbf{let!}\ y = c\ \textbf{in let!}\ x = \mathsf{async}\ \{\ \mathit{cexpr}_1\ \}\ \textbf{in}\ \mathit{cexpr}_2\ \}&\end{aligned}$$

The first equation returns 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.

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 [13]. It consists of *Map* and *Merge*, together with a unit computation. The unit computation can be used to define *Zero*. This is used only in the translation of empty computations f { () }.

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 code (assuming x_1 not free in $expr_2$ and x_2 not free in $expr_1$):

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

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

As with the earlier syntactic rules, we can leave out the non-standard aspect of the computations, 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 the parsing of *binds* in let!...and... is left-associative or right-associative.

5 Related work

Haskell and its extensions support monad comprehensions [12] and "do" notation for monads, idiom brackets [13] for applicatives and arrows [15]. These are similar to computation expressions in that they are not tied to concrete computations. However, they differ syntactically – they add multiple new notations, while computation expressions add a uniform notation resembling standard language structures. Adding arrows to computation expressions is an open question.

Python and C# generators, LINQ [14] in C# and "for" comprehensions in Scala are just a few examples of syntax for concrete computations. Although they can all be used with other computations, this is not generally considered idiomatic use. Similarly to F#, the Scala async library [21] supports loops and exception handling. However, it is implemented through full macro system.

Other encodings of effectful computations include effect handlers [17] and continuations [3]. Providing syntactic support for these may be an interesting alternative to our encoding. Interestingly, our *Run* operation resembles *reset* of delimited continuations [18] and our *Delay* is similar to *reify* of Filinsky [4].

6 Conclusions

This paper is presents a principled treatment of F# *computation expressions*. We develop a type system that captures the static semantics and relate the feature to well-known abstract computation types. Computation expressions provide a unified way for writing a wide range of computations including monoids, monads, applicative formlets and monads composed using monad transformers.

Computation expressions follow a different approach than e.g. Haskell "do" notation. They integrate a wide range of abstractions and flexibly reuse existing syntax (including loops and exception handling). The library developer can choose the appropriate syntax and use laws of abstract computations to guarantee that the computation preserves intuition about the syntax.

Such reusable syntactic extensions are becoming increasingly important. We cannot keep adding new features to support comprehensions, asynchronicity, queries and more as the "syntactic budget" is rapidly running out.

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

References

1. G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'n' play: formalizing asynchronous C#. ECOOP, 2012.
2. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. The essence of form abstraction. APLAS, 2008.
3. A. Filinski. Representing layered monads. POPL, pages 175–188, 1999.
4. A. Filinski. Monads in action. POPL, pages 483–494, 2010.
5. G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing back monad comprehensions. Haskell Symposium, pages 13–22, 2011.
6. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. PPoPP, pages 48–60, 2005.
7. HaskellWiki. Listt done right. Available at `http://www.haskell.org/haskellwiki/ListT_done_right`, 2012.
8. HaskellWiki. Monadplus. Available at `http://www.haskell.org/haskellwiki/MonadPlus`, 2012.
9. S. Krishnamurthi. Artifact evaluation for software conferences. Available at `http://cs.brown.edu/~sk/Memos/Conference-Artifact-Evaluation/`, 2012.
10. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. POPL, 1995.
11. S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electron. Notes Theor. Comput. Sci.*, 229(5), Mar. 2011.
12. C. Lüth and N. Ghani. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP, pages 133–144, 2002.
13. C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008.
14. E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. SIGMOD, pages 706–706, 2006.
15. R. Paterson. A new notation for arrows. ICFP, 2001.
16. T. Petricek. Programming with F# asynchronous sequences. Available at `http://tomasp.net/blog/async-sequences.aspx`, 2011.
17. G. Plotkin and M. Pretnar. Handlers of algebraic effects. ESOP, 2009.
18. T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. ICFP, 2009.
19. D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. PADL, 2011.
20. The F# Software Foundation. F# language specification. 2013.
21. Typesafe Inc. An asynchronous programming facility for scala. 2013.
22. P. Wadler. Monads for functional programming. In *Advanced Funct. Prog.*, 1995.