Fun with Parallel Monad Comprehensions

by Tomas Petricek ⟨tomas.petricek@cl.cam.ac.uk⟩

July 16, 2011

Monad comprehensions have an interesting history. They were the first syntactic extension for programming with monads. They were implemented in Haskell, but later replaced with plain list comprehensions and monadic `do` *notation. Now, monad comprehensions are back in Haskell, more powerful than ever before!*

Redesigned monad comprehensions generalize the syntax for working with lists. Quite interestingly, they also generalize syntax for zipping, grouping and ordering of lists. This article shows how to use some of the new expressive power when working with well-known monads. You'll learn what "parallel composition" means for parsers, a poor man's concurrency monad and an evaluation order monad.

Introduction

This article is inspired by my earlier work on *joinads* [1], an extension that adds pattern matching on abstract values to the *computation expression* syntax in F#. Computation expressions are quite similar to the `do` notation in Haskell. After implementing the F# version of joinads, I wanted to see how the ideas would look in Haskell. I was quite surprised to find out that a recent extension for GHC adds some of the expressive power of joinads to Haskell.

To add some background: the F# computation expression syntax can be used to work with *monads*, but also with *monoids* and a few other abstract notions of computation. It also adds several constructs that generalize imperative features of F#, including `while` and `for` loops as well as exception handling. The *joinads* extension adds support for pattern-matching on "monadic values". For example, you can define a parallel programming monad and use joinads to wait until two

parallel computations both complete or wait until the first of the two completes returning a value matching a particular pattern.

How are F# joinads related to Haskell? A recent GHC patch implemented by Nils Schweinsberg [2, 3] brings back support for monad comprehensions to Haskell. The change is now a part of the main branch and will be available in GHC starting with the 7.2 release. The patch doesn't just re-implement original monad comprehensions, but also generalizes recent additions to list comprehensions, allowing parallel monad comprehensions and monadic versions of operations like ordering and grouping [4].

The operation that generalizes parallel comprehensions is closely related to a `merge` operation that I designed for F# joinads. In the rest of this article, I demonstrate some of the interesting programs that can be written using this operation and the elegant syntax provided by the re-designed monad comprehensions.

Quick review of list comprehensions

List comprehensions are a very powerful mechanism for working with lists in Haskell. I expect that you're already familiar with them, but let me start with a few examples. I will use the examples later to demonstrate how the generalized monad comprehension syntax works in a few interesting cases.

If we have a list `animals` containing "cat" and "dog" and a list `sounds` containing animal sounds "meow" and "woof", we can write the following snippets:

```
> [ a ++ " " ++ s | a <- animals, s <- sounds ]
["cat meow","cat woof","dog meow","dog woof"]

> [ a ++ " " ++ s | a <- animals, s <- sounds, a !! 1 == s !! 1 ]
["dog woof"]

> [ a ++ " " ++ s | a <- animals | s <- sounds ]
["cat meow","dog woof"]
```

The first example uses just the basic list comprehension syntax. It uses two *generators* to implement a Cartesian product of the two collections. The second example adds a guard to specify that we want only pairs of strings whose second character is the same. The guard serves as an additional filter for the results.

The last example uses parallel list comprehensions. The syntax is available after enabling the ParallelListComp language extension. It allows us to take elements from multiple lists, so that the n^{th} element of the first list is matched with the n^{th} element of the second list. The same functionality can be easily expressed using the `zip` function.

Generalizing to monad comprehensions

The three examples we've seen in the previous section are straightforward when working with lists. After installing the latest development snapshot of GHC and turning on the MonadComprehensions language extension, we can use the same syntax for working with further notions of computation. If we instantiate the appropriate type classes, we can even use guards, parallel comprehensions and operations like ordering or grouping. Figure 1 shows some of the type classes and functions that are used by the desugaring.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a

class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

class (Monad m) => MonadZip m where
  mzip :: m a -> m b -> m (a, b)

guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else mzero
```

Figure 1: Type classes and functions used by monad comprehensions

Aside from Monad, the desugaring also uses the MonadPlus and MonadZip type classes. The former is used only for the `guard` function, which is also defined in Figure 1. The latter is a new class which has been added as a generalization of parallel list comprehensions. The name of the function makes it clear that the type class is a generalization of the `zip` function. The patch also defines a MonadGroup type class that generalizes grouping operations inside list comprehensions, but I will not discuss that feature in this article.

You can find the general desugaring rules in the patch description [2]. In this article, we'll just go through the examples from the previous section and examine what the translation looks like. The following declaration shows how to implement the Monad, MondPlus, and MonadZip type classes for lists:

```
instance Monad [] where
  source >>= f = concat $ map f source
  return a = [a]
```

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)

instance MonadZip [] where
  mzip = zip
```

The >>= operation, called bind, applies the provided function to each element of the input list and then concatenates the generated lists. The `return` function creates a singleton list containing the specified value. The `mzero` value from MonadPlus type class is an empty list, which means that `guard` returns `[()]` when the argument is `True` and the empty list otherwise. Finally, the `mzip` function for lists is just `zip`.

Now we have everything we need to look at the desugaring of monad comprehensions. The first example from the previous section used multiple generators and can be translated purely in terms of Monad:

```
animals >>= (\a -> sounds >>= (\s ->
  return $ a ++ " " ++ b))
```

Every generator is translated to a binding using >>=. The operations are nested, and the innermost operation always returns the result of the output function. The next snippet shows what happens when we add a predicate to filter the results:

```
animals >>= (\a -> sounds >>= (\s ->
  guard (a !! 1 == s !! 1) >>= (\_ ->
    return $ a ++ " " ++ s) ))
```

A predicate is translated into a call to the `guard` function in the innermost part of the desugared expression. When the function returns `mzero` value (an empty list), the result of the binding will also be `mzero`, so the element for which the predicate doesn't hold will be filtered out. Finally, let's look at the translation of the last example:

```
(animals `mzip` sounds) >>= (\(a, s) ->
  return $ a ++ " " ++ s)
```

When we use parallel comprehensions, the inputs of the generators are combined using the `mzip` function. The result is passed to the bind operation, which applies the output function to values of the combined computation. If we also specified filtering, the `guard` function would be added to the innermost expression, as in the previous example.

As you can see, the translation of monad comprehensions is quite simple, but it adds expressivity to the syntax for working with monads. In particular, the `do` notation doesn't provide an equivalent syntactic extension for writing parallel comprehensions. (Constructs like generalized ordering, using functions of type `m a -> m a`, and generalized grouping, using functions of type `m a -> m (m a)`, add even more expressivity, but that's a topic for another article.) In the next three sections, I show how we could implement the `mzip` operation for several interesting monads, representing parsers, resumable computations, and parallel computations. At the end of the article, I also briefly consider laws that should hold about the `mzip` operation.

Composing parsers in parallel

What does a *parallel composition of two parsers* mean? Probably the best thing we can do is to run both parsers on the input string and return a tuple with the two results. That sounds quite simple, but what is this construct good for? Let's first implement it and then look at some examples.

Introducing parsers

A parser is a function that takes an input string and returns a list of possible results. It may be empty (if the parser fails) or contain several items (if there are multiple ways to parse the input). The implementation I use in this article mostly follows the one by Hutton and Meijer [5].

```
newtype Parser a
  = Parser (String -> [(a, Int, String)])
```

The result of parsing is a tuple containing a value of type `a` produced by the parser, the number of characters consumed by the parser, and the remaining unparsed part of the string. The `Int` value represents the number of characters consumed by the parser. It is not usually included in the definition, but we'll need it in the implementation of `mzip`.

Now that we have a definition of parsers, we can create our first primitive parser and a function that runs a parser on an input string and returns the results:

```
item :: Parser Char
item = Parser (\input -> case input of
  ""   -> []
  c:cs -> [(c, 1, cs)])
```

```
run :: Parser a -> [a]
run (Parser p) input =
  [ result | (result, _, tail) <- p input, tail == [] ]
```

The `item` parser returns the first character of the input string. When it succeeds, it consumes a single character, so it returns 1 as the second element of the tuple. The `run` function applies the underlying function of the parser to a specified input. As specified by the condition `tail == []`, the function returns the results of those parsers which parsed the entire input. The next step is to make the parser monadic.

Implementing the parser monad

Parsers are well known examples of *monads* and of *monoids*. This means that we can implement both the Monad and the MonadPlus type classes for our `Parser` type. You can find the implementation in Figure 2.

```
instance Monad Parser where
  return a = Parser (\input -> [(a, 0, input)])
  (Parser p1) >>= f = Parser (\input ->
    [ (result, n1 + n2, tail)
        | (a, n1, input') <- p1 input
        , let (Parser p2) = f a
        , (result, n2, tail) <- p2 input' ])

instance MonadPlus Parser where
  mzero = Parser (\_ -> [])
  mplus (Parser p1) (Parser p2) = Parser (\input ->
    p1 input ++ p2 input)
```

Figure 2: Instances of Monad and MonadPlus for parsers

The `return` operation returns a single result containing the specified value that doesn't consume any input. The `>>=` operation can be implemented using ordinary list comprehensions. It runs the parsers in sequence, returns the result of the second parser and consumes the sum of characters consumed by the first and the second parser. The `mzero` operation creates a parser that always fails, and `mplus` represents a nondeterministic choice between two parsers.

The two type class instances allow us to use some of the monad comprehension syntax. We can now use the `item` primitive to write a few simple parsers:

```
sat :: (Char -> Bool) -> Parser Char
```

```
sat pred   = [ ch | ch <- item, pred ch ]

char, notChar :: Char -> Parser Char
char ch    = sat (ch ==)
notChar ch = sat (ch /=)

some p = [ a:as | a <- p, as <- many p ]
many p = some p `mplus` return []
```

The `sat` function creates a parser that parses a character matching the specified predicate. The *generator syntax* `ch <- item` corresponds to monadic binding and is desugared into an application of the `>>=` operation. Because the Parser type is an instance of MonadPlus, we can use the predicate `pred ch` as a guard. The desugared version of the function is:

```
sat pred = item >>= (\ch ->
  guard (pred ch) >>= (\_ -> return ch))
```

The `some` and `many` combinators are mutually recursive. The first creates a parser that parses one or more occurrences of `p`. We encode it using a monad comprehension with two bindings. The parser parses `p` followed by `many p`. Another way to write the `some` parser would be to use combinators for working with applicative functors. This would allow us to write just `(:) <$> p <*> many p`. However, using combinators becomes more difficult when we need to specify a guard as in the `sat` parser. Monad comprehensions provide a uniform and succinct alternative.

The order of monadic bindings usually matters. The monad comprehension syntax makes this fact perhaps slightly less obvious than the `do` notation. To demonstrate this, let's look at a parser that parses the body of an expression enclosed in brackets:

```
brackets :: Char -> Char -> Parser a -> Parser a
brackets op cl body =
  [ inner
      | _ <- char op
      , inner <- brackets op cl body `mplus` body
      , _ <- char cl ]

skipBrackets = brackets '(' ')' (many item)
```

The `brackets` combinator takes characters representing opening and closing brackets and a parser for parsing the body inside the brackets. It uses a monad comprehension with three binding expressions that parse an opening brace, the body or more brackets, and then the closing brace.

If you run the parser using `run skipBrackets "((42))"` you get a list containing `"42"`, but also `"(42)"`. This is because the `many item` parser can also consume brackets. To correct that, we need to write a parser that accepts any character except opening and closing brace. As we will see shortly, this can be elegantly solved using parallel comprehensions.

Parallel composition of parsers

To support parallel monad comprehensions, we need to implement MonadZip. As a reminder, the type class defines an operation `mzip` with the following type:

```
mzip :: m a -> m b -> m (a, b)
```

By looking just at the type signature, you can see that the operation can be implemented in terms of `>>=` and `return` like this:

```
mzip ma mb = ma >>= \a -> mb >>= \b -> return (a, b)
```

This is a reasonable definition for some monads, such as the `Reader` monad, but not for all of them. For example, `mzip` for lists should be `zip`, but the definition above would behave as a Cartesian product! A more interesting definition for parsers, which cannot be expressed using other monad primitives, is parallel composition from Figure 3.

```
instance MonadZip Parser where
  mzip (Parser p1) (Parser p2) = Parser (\input ->
    [ ((a, b), n1, tail1)
        | (a, n1, tail1) <- p1 input
        , (b, n2, tail2) <- p2 input
        , n1 == n2 ])
```

Figure 3: Instance of MonadZip type class for parsers

The parser created by `mzip` independently parses the input string using both of the parsers. It uses list comprehensions to find all combinations of results such that the number of characters consumed by the two parsers was the same. For each matching combination, the parser returns a tuple with the two parsing results. Requiring that the two parsers consume the same number of characters is not an arbitrary decision. It means that the remaining unconsumed strings `tail1` and `tail2` are the same and so we can return either of them. Using a counter is more efficient than comparing strings and it also enables working with infinite strings.

Let's get back to the example with parsing brackets. The following snippet uses parallel monad comprehensions to create a version that consumes all brackets:

```
skipAllBrackets = brackets '(' ')' body
  where body = many [ c | c <- notChar '(' | _ <- notChar ')' ]
```

The parser `body` takes zero or more of any characters that are not opening or closing brackets. The parallel comprehension runs two `notChar` parsers on the same input. They both read a single character and they succeed if the character is not '(' and ')' respectively. The resulting parser succeeds only if both of them succeed. Both parsers return the same character, so we return the first one as the result and ignore the second.

Another example where this syntax is useful is validation of inputs. For example, a valid Cambridge phone number consists of 10 symbols, contains only digits, and starts with 1223. The new syntax allows us to directly encode these three rules:

```
cambridgePhone =
  [ n | n <- many (sat isDigit)
      | _ <- replicateM 10 item
      | _ <- startsWith (string "1223") ]
```

The encoding is quite straightforward. We need some additional combinators, such as `replicateM`, which repeats a parser a specified number of times, and `startsWith`, which runs a parser and then consumes any number of characters.

We could construct a single parser that recognizes valid Cambridge phone numbers without using `mzip`. The point of this example is that we can quite nicely combine several independent rules, which makes the validation code easy to understand and extend.

Parallel composition of context-free parsers

Monadic parser combinators are very expressive. In fact, they are often *too* expressive, which makes it difficult to implement the combinators efficiently. This was a motivation for the development of non-monadic parsers, such as the one by Swierstra [6, 7], which are less expressive but more efficient. *Applicative functors*, developed by McBride and Paterson [8], are a weaker abstraction that can be used for writing parsers. Figure 4 shows the Haskell type class Applicative that represents applicative functors.

If you're familiar with applicative functors, you may know that there is an alternative definition of Applicative that uses an operation with the same type signature as `mzip`. We could use `mzip` to define an Applicative instance, but this would give us a very different parser definition! In some sense, the following example combines two different applicative functors, but I'll write more about that in the next section.

```
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Figure 4: Definition of the Applicative type class

The usual applicative parsers allow us to write parsers where the choice of the next parser doesn't depend on the value parsed so far. In terms of formal language theory, they can express only *context-free* languages. This is still sufficient for many practical purposes. For example, our earlier `brackets` parser can be written using the applicative combinators:

```
brackets op cl body =
  pure (\_ inner _ -> inner)
    <*> char op
    <*> brackets op cl body `mplus` body
    <*> char cl
```

The example first creates a parser that always succeeds and returns a function using the `pure` combinator. Then it applies this function (contained in a parser) to three arguments (produced by the three parsers). The details are not important, but the example shows that comprehensions with independent generators can be expressed just using the Applicative interface.

The interesting question is, what operation does `mzip` represent for context-free grammars? A language we obtain if parses for two other languages both succeed is an *intersection* of the two languages. An intersection of two context-free languages is not necessarily context-free, which can be demonstrated using the following example:

$$A = \{a^m b^m c^n | m, n \geq 0\} \qquad B = \{a^n b^m c^m | m, n \geq 0\}$$
$$A \cap B = \{a^m b^m c^m | m \geq 0\}$$

The language A accepts words that start with some number of 'a' followed by the same number of 'b' and then arbitrary number of 'c' symbols. The language B is similar, but it starts with a group of any length followed by two groups of the same length. Both are context-free. In fact, our parser `brackets` can be used to parse the two character groups with the same length.

The intersection $A \cap B$ is not context-free [9], but we can easily encode it using parallel composition of parsers. We don't need the full power of monads to parse the first group and calculate its length. It could be implemented just in terms of Applicative and `mzip`, but we use the nicer monad comprehension syntax:

```
[ True | _ <- many $ char 'a', _ <- brackets 'b' 'c' unit
       | _ <- brackets 'a' 'b' unit, _ <- many $ char 'c' ]
  where unit = return ()
```

The example uses both parallel and sequential binding, but the sequential composition doesn't involve dependencies on previously parsed results. It uses `brackets` to parse two groups of the same length, followed (or preceded) by `many` to consume the remaining group of arbitrary length.

Applicative and parallel parsers

Before moving to the next example, let's take a look how `mzip` relates to applicative functors. As already mentioned, an alternative definition of applicative functors ([8] section 7) uses an operation with exactly the same type signature as `mzip`. You can find the alternative definition in Figure 5.

```
class Functor f => Monoidal f where
  unit  :: f ()
  (⋆)   :: f a -> f b -> f (a, b)

pure :: Monoidal f => a -> f a
pure a = fmap (const a) unit

(<*>) :: Monoidal f => f (a -> b) -> f a -> f b
f <*> a = fmap (uncurry ($)) (f ⋆ a)
```

Figure 5: Alternative definition of Applicative operations

Applicative functors are more general than monads, which means that every monad is also an applicative functor. When introducing the `mzip` operation, we attempted to define it in terms of `return` and `>>=`. That code was actually a definition of $\star$. However, as already explained, we cannot use that definition (or the $\star$ operation) for `mzip`, because that definition isn't always intuitively right. Referring to intuition is always tricky, so I'll express my expectations more formally in terms of laws at the end of the article. However, if we always just used $\star$ as the `mzip` operation, we wouldn't get any additional expressive power, so there would be no point in using parallel monad comprehensions. The expression `[ e | a <- e1 | b <- e2]` would mean exactly the same thing as `[ e | a <- e1, b <- e2]`.

For example, the definition of $\star$ for the usual List monad gives us the Cartesian product of lists. This isn't very useful, because we can get that behavior using

multiple generators. Instead, parallel list comprehensions use zipping of lists, which comes from a different applicative functor, namely ZipList.

The example with parsers is similar. The implementation of $\star$ would give us sequential composition of parsers. This wouldn't be very useful, so I defined `mzip` as the intersection of parsers. This is an interesting operation that adds expressive power to the language. The `mzip` operation also defines an instance of applicative functor for parsers, but a different one.

For the fans of category theory, the $\star$ operation is a natural transformation of a *monoidal functor* that can be defined by the monad we're working with. The `mzip` operation can be also viewed as a natural transformation of some monoidal functor, but it may be a different one. One of the additional laws that we can require about `mzip` is commutativity (more about that later), which means that `mzip` should be defined by a *symmetric monoidal functor*.

Parallelizing cooperative computations

As the name *parallel* monad comprehensions suggests, we can use the syntax for running computations in parallel. Unlike comprehensions with multiple generators, parallel comprehensions cannot have any dependencies between the composed bindings. This means that the computations can be evaluated independently.

In this section, I demonstrate the idea using a poor man's concurrency monad inspired by Claessen [10]. The monad can be used to implement a lightweight cooperative concurrency. When running two computations in parallel, we can allow interleaving of atomic actions from the two threads.

Modelling resumable computations

The example I demonstrate here models computations using *resumptions*. This concept is slightly simpler than the original poor man's concurrency monad (which is based on continuations). A resumption is a computation that has either finished and produced some value or has not finished, in which case it can run one atomic step and produce a new resumption:

```
data Monad m => Resumption m r
  = Step (m (Resumption m r))
  | Done r
```

The type is parametrized over a monad and a return type. When evaluating a resumption, we repeatedly run the computation step-by-step. While evaluating, we perform the effects allowed by the monad `m` until we eventually get a result of type `r`. If you're interested in more details about the `Resumption` type, you can find

similar definitions in Harrison's cheap threads [11] and Papaspyrou's resumption transformer [12].

Now that we have a type, we can write a function to create a `Resumption` that runs a single atomic action and then completes, and a function that runs a `Resumption`:

```
run :: Monad m => Resumption m r -> m r
run (Done r) = return r
run (Step m) = m >>= run

action :: Monad m => m r -> Resumption m r
action a = Step [ Done r | r <- a ]
```

The `run` function takes a resumption that may perform effects specified by the monad `m` and runs the resumption inside the monad until it reaches `Done`. The function runs the whole computation sequentially (and it cannot be implemented differently). The cooperative concurrency can be added later by creating a combinator that interleaves the steps of two `Resumption` computations.

The `action` function is quite simple. It returns a `Step` that runs the specified action inside the monad `m` and then wraps the result using the `Done` constructor. I implemented the function using monad comprehensions to demonstrate the notation again, but it could be equally written using combinators or the `do` notation.

Implementing the resumption monad

You can see the implementation of Monad instance for `Resumption m` in Figure 6. The `return` operation creates a new resumption in the "done" state which contains the specified value. The `>>=` operation constructs a resumption that gets the result of the first resumption and then calls the function `f`. When the left parameter is `Done`, we apply the function to the result and wrap the application inside `return` (because the function is pure) and `Step`. When the left parameter is `Step`, we create a resumption that runs the step and then uses `>>=` recursively to continue running steps from the left argument until it finishes.

The listing also defines an instance of MonadTrans to make `Resumption` a monad transformer. The `lift` function takes a computation in the monad `m` and turns it into a computation in the `Resumption` monad. This is exactly what our function for wrapping atomic actions does.

Equipped with the two type class instances and the `run` function, we can write some interesting computations. The following function creates a computation that runs for the specified number of steps, prints some string in each step and then returns a specified value:

```
instance Monad m => Monad (Resumption m) where
  return a = Done a
  (Done r) >>= f = Step $ return (f r)
  (Step s) >>= f = Step $ do
    next <- s
    return $ next >>= f

instance MonadTrans Resumption where
  lift = action
```

Figure 6: Instances of Monad and MonadTrans for resumptions.

```
printLoop :: String -> Int -> a -> Resumption IO a
printLoop str count result = do
  lift $ putStrLn str
  if count == 1 then return result
  else printLoop str (count - 1) result

cats = run $ printLoop "meow" 3 "cat"
```

The function is written using the `do` notation. It first prints the specified string, using `lift` to turn the `IO ()` action into a single-step `Resumption IO ()`. When the counter reaches one, it returns the result; otherwise it continues looping.

The snippet defines a simple computation `cats` that prints "meow" three times and then returns a string "cat". The computations created by `printLoop` are not fully opaque. If we have two computations like `cats`, we can treat them as sequences of steps and interleave them. This is what the `mzip` operation does.

Parallel composition of resumptions

If we have two resumptions, we can compose them to run in sequence either using the `do` notation or using a monad comprehension with two generators. To run them in parallel, we need to implement interleaving of the steps as shown in Figure 7.

The result of `mzip` is a resumption that consists of multiple steps. In each step, it performs one step of both of the resumptions given as arguments. When it reaches a state when both of the resumptions complete and produce results, it returns a tuple containing the results using `Done`. A step is performed using an effectful `step` function. To keep the implementation simple, we keep applying `step` to both of the resumptions and then recursively combine the results. Applying `step` to a

```
instance Monad m => MonadZip (Resumption m) where
  mzip (Done a) (Done b) = Done (a, b)
  mzip sa sb = Step [ mzip a b | a <- step sa, b <- step sb ]
    where step (Done r) = return $ Done r
          step (Step sa) = sa
```

Figure 7: Instance of MonadZip that composes resumptions in parallel

resumption that has already completed isn't a mistake. This operation doesn't do anything and just returns the original resumption (without performing any effects).

Once we define `mzip`, we can start using the parallel comprehension syntax for working with resumptions. The next snippet demonstrates two ways of composing resumptions. In both examples, we compose a computation that prints "meow" two times and then returns "cat" with a computation that prints "woof" three times and then returns "dog":

```
animalsSeq =
  [ c ++ " and " ++ d
    | c <- printLoop "meow" 2 "cat"
    , d <- printLoop "woof" 3 "dog" ]

animalsPar =
  [ c ++ " and " ++ d
    | c <- printLoop "meow" 2 "cat"
    | d <- printLoop "woof" 3 "dog" ]
```

The only difference between the two examples is that the first one composes the operations using multiple generators (separated by comma) and the second one uses parallel comprehensions (separated by bar).

When you run the first example, the program prints meow, meow, woof, woof, woof and then returns a string "cat and dog". The second program interleaves the steps of the two computations and prints meow, woof, meow, woof, woof and then returns the same string.

Composing computations in parallel

In the previous section, we used the parallel comprehension syntax to create computations that model parallelism using resumptions. Resumptions can be viewed as lightweight cooperative threads. They are useful abstraction, but the simple implementation in the previous section does not give us any speed-up on multi-core

CPU. This section will follow a similar approach, but we look at how to implement actual parallelism based on *evaluation strategies*.

Marlow *et al.* [13] introduced an Eval monad for explicitly specifying evaluation order. I will start by briefly introducing the monad, so don't worry if you're not familiar with it already. I'll then demonstrate how to define a MonadZip instance for this monad. This way, we can use the parallel comprehension syntax for actually running computations in parallel.

Introducing the evaluation-order monad

The evaluation-order monad is represented by a type `Eval a`. When writing code inside the monad, we can use several functions of type `a -> Eval a` that are called *strategies*. These functions take a value of type `a`, which may be unevaluated, and wrap it inside the monad. A strategy can specify how to evaluate the (unevaluated) value. The two most common strategies are `rpar` and `rseq` (both are functions of type `a -> Eval a`). The `rpar` strategy starts evaluating the value in background and `rseq` evaluates the value eagerly before returning.

A typical pattern is to use the `do` notation to spawn one computation in parallel and then run another computation sequentially. This way we can easily parallelize two function calls:

```
fib38 = runEval $ do
  a <- rpar $ fib 36
  b <- rseq $ fib 37
  return $ a + b
```

The example shows how to calculate the 38^{th} Fibonacci number. It starts calculating `fib 36` in parallel with the rest of the computation and then calculates `fib 37` sequentially. The `do` block creates a value of type `Eval Integer`. We then pass this value to `runEval`, which returns the wrapped value. Because we used the `rpar` and `rseq` combinators when constructing the computation, the returned value will be already evaluated.

However, it is worth noting that the `return` operation of the monad doesn't specify any evaluation order. The function `runEval . return` is just an identity function that doesn't force evaluation of the argument. The evaluation order is specified by additional combinators such as `rpar`.

The Eval monad is implemented in the parallel package [14] and very well explained in the paper by Marlow *et al.* [13]. You can find the definition of `Eval` and its monad instance in Figure 8. We don't need to know how `rpar` and `rseq` work, so we omit them from the listing.

The `Eval a` type is simple. It just wraps a value of type `a`. The `runEval` function unwraps the value and the Monad instance implements composition of

```
data Eval a = Done a

runEval :: Eval a -> a
runEval (Done x) = x

instance Monad Eval where
  return x = Done x
  Done x >>= k = k x
```

Figure 8: Evaluation-order monad Eval with a Monad instance

computations in the usual way. The power of the monad comes from the evaluation annotations we can add. We transform values of type `a` into values of type `Eval a`; while doing so, we can specify the evaluation strategy. The strategy is usually given using combinators, but if we add an instance of the MonadZip class, we can also specify the evaluation order using the monad comprehension syntax.

Specifying parallel evaluation order

The `mzip` operator for the evaluation order monad encapsulates a common pattern that runs two computations in parallel. We've seen an example in the previous section – the first computation is started using `rpar` and the second one is evaluated eagerly in parallel using `rseq`. You can find the implementation in Figure 9.

```
instance MonadZip Eval where
  mzip ea eb = do
    a <- rpar $ runEval ea
    b <- rseq $ runEval eb
    return (a, b)
```

Figure 9: Instance of MonadZip for parallelizing tasks

A tricky aspect of `Eval` is that it may represent computations with explicitly specified evaluation order (created, for example, using `rpar`). We can also create computations without specifying evaluation order using `return`. The fact that `return` doesn't evaluate the values makes it possible to implement the `mzip` function, because the required type signature is `Eval a -> Eval b -> Eval (a, b)`.

The two arguments already have to be values of type `Eval`, but we want to specify the evaluation order using `mzip` after creating them. If all `Eval` values were already

evaluated, then the `mzip` operation couldn't have any effect. However, if we create values using `return`, we can then apply `mzip` and specify how they should be evaluated later. This means that `mzip` only works for `Eval` computations created using `return`.

Once we understand this, implementing `mzip` is quite simple. It extracts the underlying (unevaluated) values using `runEval`, specifies the evaluation order using `rpar` and `rseq` and then returns a result of type `Eval (a, b)`, which now carries the evaluation order specification. Let's look how we can write a sequential and parallel version of a snippet that calculates the 38^{th} Fibonacci number:

```
fibTask n = return $ fib n

fib38seq = runEval [ a + b | a <- fibTask 36
                           , b <- fibTask 37 ]
fib38par = runEval [ a + b | a <- fibTask 36
                           | b <- fibTask 37 ]
```

The snippet first declares a helper function `fibTask` that creates a delayed value using the sequential `fib` function and wraps it inside the `Eval` monad without specifying evaluation strategy. Then we can use the function as a source for generators in the monad comprehension syntax. The first example runs the entire computation sequentially – aside from some wrapping and unwrapping, there are no evaluation order specifications. The second example runs the two sub-computations in parallel. The evaluation order annotations are added by the `mzip` function from the desugared parallel comprehension syntax.

To run the program using multiple threads, you need to compile it using GHC with the `-threaded` option. Then you can run the resulting application with command line arguments `+RTS -N2 -RTS`, which specifies that the runtime should use two threads. I measured the performance on a dual-core Intel Core 2 Duo CPU (2.26GHz). The time needed to run the first version was approximately 13 seconds while the second version completes in 9 seconds.

Writing parallel algorithms

The $\sim$1.4$\times$ speedup is less than the maximal 2$\times$ speedup, because the example parallelizes two calculations that do not take equally long. To generate a better potential for parallelism, we can implement a recursive `pfib` function that splits the computation into two parallel branches recursively until it reaches some threshold:

```
pfib :: Integer -> Eval Integer
pfib n | n <= 35 = return $ fib n
```

```
pfib n = [ a + b | a <- pfib $ n - 1
                 | b <- pfib $ n - 2 ]
```

I hope you'll agree that the declaration looks quite neat. A nice consequence of using parallel comprehensions is that we can see which parts of the computation will run in parallel without any syntactic noise. We just replace a comma with a bar to get a parallel version! The compiler also prevents us from trying to parallelize code that cannot run in parallel, because of data dependencies. For example, let's look at the Ackermann function:

```
ack :: Integer -> Integer -> Eval Integer
ack 0 n = return $ n + 1
ack m 0 = ack (m - 1) 1
ack m n = [ a | na <- ack m (n - 1)
              , a <- ack (m - 1) na ]
```

The Ackermann function is a well-known function from computability theory. It is interesting because it grows very fast (as a result, it cannot be expressed using primitive recursion). For example, the value of `ack 4 2` is $2^{65536} - 3$.

We're probably not going to be able to finish the calculation, no matter how many cores our CPU has. However, we can still try to parallelize the function by replacing the two sequential generators with a parallel comprehension:

```
ack m n = [ a | na <- ack m (n - 1)
              | a <- ack (m - 1) na ]
```

If you try compiling this snippet, you get an error message saying `Not in scope: na`. We can easily see what went wrong if we look at the desugared version:

```
((ack m (n - 1)) `mzip`
    (ack (m - 1) na)) >>= (\(na, a) -> a)
```

The problem is that the second argument to `mzip` attempts to access the value `na` which is defined later. The value is the result of the first expression, so we can access it only after both of the two parallelized operations complete.

In other words, there is a data dependency between the computations that we're trying to parallelize. If we were not using parallel monad comprehensions, we could mistakenly think that we can parallelize the function and write the following:

```
ack m n = runEval $ do
  na <- rpar $ ack m (n - 1)
  a  <- rseq $ ack (m - 1) na
  return a
```

This would compile, but it wouldn't run in parallel! The value `na` needs to be evaluated before the second call, so the second call to `ack` will block until the first one completes. This demonstrates a nice aspect of writing parallel computations using the comprehension syntax. Not only that the syntax is elegant, but the desugaring also performs a simple sanity check on our code.

Parallel comprehension laws

I have intentionally postponed the discussion about laws to the end of the article. So far, we have looked at three different implementations of `mzip`. I discussed some of the expectations informally to aid intuition. The type of `mzip` partially specifies how the operation should behave, but not all well-typed implementations are intuitively right.

In my understanding, the laws about `mzip` are still subject to discussion, although some were already proposed in the discussion about the GHC patch [2]. I hope to contribute to the discussion in this section. We first look at the laws that can be motivated by the category theory behind the operation, and then discuss additional laws inspired by the work on F# joinads.

Basic laws of parallel bindings

As already briefly mentioned, the `mzip` operation can be viewed as a *natural transformation* defined by some *monoidal functor* [15]. In practice, this means that the `mzip` operation should obey two laws. The first one is usually called *naturality* and it specifies the behavior of `mzip` with respect to the `map` function of a monad (the function can be implemented in terms of bind and return and corresponds to `liftM` from the Haskell base library). The second law is *associativity*, and we can express it using a helper function $assoc\ ((a, b), c) = (a, (b, c))$:

$$map\ (f \times g)\ (mzip\ a\ b) \equiv mzip\ (map\ f\ a)\ (map\ g\ b) \tag{1}$$

$$mzip\ a\ (mzip\ b\ c) \equiv map\ assoc\ (mzip\ (mzip\ a\ b)\ c) \tag{2}$$

The naturality law (1) specifies that we can change the order of applying `mzip` and `map`. The equation has already been identified as a law in the discussion about the patch [2]. The law is also required by *applicative functors* [8] – this is not surprising as applicative functors are also monoidal.

The *associativity* law (2) is also very desirable. When we write a comprehension such as `[e | a <- m1 | b <- m2 | c <- m3]`, the desugaring first needs to zip two of the three inputs and then zip the third with the result, because `mzip` is a

binary operation. The order of zipping feels like an implementation detail, but if we don't require associativity, it may affect the meaning of our code.

Parallel binding as monoidal functor

A monoidal functor in category theory defines a natural transformation (corresponding to our `mzip`), but also a special value called *unit*. A Haskell representation of monoidal functor is the Monoidal type class from Figure 5. For a monoidal functor `f`, the type of unit is `f ()`. Every applicative functor in Haskell has a unit. For example, the unit value for ZipList is an infinite list of `()` values. We do not necessarily need to add unit to the definition of MonadZip, because it is not needed by the desugaring. However, it is interesting to explore how a special `munit` value would behave.

The laws for monoidal functors specify that if we combine `munit` with any other value using `mzip`, we can recover the original value using `map snd`. In the language of monad comprehensions, the law says that the expression `[ e | a <- m ]` should be equivalent to `[ e | a <- m | () <- munit]`.

It is important to realize that the computation created using monadic `return` isn't the same thing as unit of the monoidal functor associated with `mzip`. For lists, `return ()` creates a singleton list. The `return` operation is unit of a monoidal functor defined by the monad, but the unit associated with `mzip` belongs to a different monoidal functor!

Symmetry of parallel binding

Another sensible requirement for `mzip` (which exists in F# joinads) is that reordering of the arguments only changes the order of elements in the resulting tuple. In theory, this means that the *monoidal functor* defining `mzip` is *symmetric*. We can specify the law using a helper function $swap\ (a, b) = (b, a)$:

$$mzip\ a\ b \equiv map\ swap\ (mzip\ a\ b) \tag{3}$$

The *symmetry* law (3) specifies that `[(x, y) | x <- a | y <- b]` should mean the same thing as `[(x, y) | y <- b | x <- a]`. This may look like a very strong requirement, but it fits quite well with the usual intuition about the `zip` operation and parallel monad comprehensions. The symmetry law holds for lists, parsers and the evaluation order monad. For the poor man's concurrency monad, it holds if we treat effects that occur within a single step of the evaluation as unordered (which may be a reasonable interpretation). For some monads, such as the `State` monad, it is not possible to define a symmetric `mzip` operation.

The three laws that we've seen so far are motivated by the category theory laws for the *(symmetric) monoidal functors* that define our `mzip` operation. However,

we also need to relate the functors in some way to the monads with which we are combining them.

Relations with additional operations

The discussion about the patch [2] suggests one more law that relates the `mzip` operation with the `map` operation of the monad, called *information preservation*:

$$map\ fst\ (mzip\ a\ b) \equiv a \equiv map\ snd\ (mzip\ b\ a) \tag{4}$$

The law specifies that combining a computation with some other computation using `mzip` and then recovering the original form of the value using `map` doesn't lose information. However, this law is a bit tricky. For example, it doesn't hold for lists if a and b are lists of different length, since the `zip` function restricts the length of the result to the length of the shorter list. Similarly, it doesn't hold for parsers (from the first section) that consume a different number of characters.

However, the law expresses an important requirement: when we combine certain computations, it should be possible to recover the original components. A law that is similar to (4) holds for applicative functors, but with a slight twist:

$$map\ fst\ (mzip\ a\ munit) \equiv a \equiv map\ snd\ (mzip\ munit\ a) \tag{5}$$

Instead of zipping two arbitrary monadic values, the law for applicative functors zips an arbitrary value with *unit*. In case of lists, unit is an infinite list, so the law holds. Intuitively, this holds because *unit* has a maximal structure (in this case, the structure is the length of the list).

Ideally, we'd like to say that combining two values with the same structure creates a new value which also has the same structure. There is no way to refer to the "structure" of a monadic value directly, but we can create values with a given structure using `map`. This weaker law holds for both lists and parsers. Additionally, we can describe the case when one of the two computations is `mzero` and when both of them are created using `return`:

$$map\ fst\ (mzip\ a\ (map\ f\ a)) \equiv a \equiv map\ snd\ (mzip\ (map\ g\ a)\ a) \tag{6}$$

$$map\ fst\ (mzip\ a\ mzero) \equiv mzero \equiv map\ snd\ (mzip\ mzero\ a) \tag{7}$$

$$mzip\ (return\ a)\ (return\ b) \equiv return\ (a, b) \tag{8}$$

The first law (6) is quite similar to the original law (4). The only difference is that instead of zipping with an arbitrary monadic value b, we're zipping a with a value constructed using `map` (for any total functions f and g). This means that the

actual value(s) that the monadic value contains (or produces) can be different, but the structure will be the same, because `map` is required to preserve the structure.

The second law (7) specifies that `mzero` is the *zero element* with respect to `mzip`. It almost seems that it holds of necessity, because `mzero` with type `m a` doesn't contain any value of type `a`, so there is no way we could construct a value of type `(a,b)`. This second law complements the first (6), but it is not difficult to see that it contradicts the original *information preservation* law (4).

Finally, the third law (8) describes how `mzip` works with respect to the monadic `return` operation. This is interesting, because we're relating `mzip` of one applicative functor with the unit of another applicative functor (defined by the monad). The law isn't completely arbitrary; an equation with a similar structure is required for *causal commutative arrows* [16]. In terms of monad comprehension syntax, the law says that `[ e | a <- return e1 | b <- return e2 ]` is equivalent to `[ e | (a, b) <- return (e1, e2)]`.

I believe that the three laws I proposed in this section partly answer the question of how to relate the two structures combined by parallel monad comprehensions – the MonadPlus type class with the symmetric monoidal functor that defines `mzip`.

Conclusions

After some time, monad comprehensions are back in Haskell! The recent GHC patch makes them even more useful by generalizing additional features of list comprehensions including parallel binding and support for operations like ordering and grouping. In this article, I focused on the first generalization, although the remaining two are equally interesting.

We looked at three examples: parallel composition of parsers (which applies parsers to the same input), parallel composition of resumptions (which interleaves the steps of computations) and parallel composition of an evaluation order monad (which runs two computations in parallel). Some of the examples are inspired by my previous work on joinads that add a similar language extension to F#. The F# version includes an operation very similar to `mzip` from the newly included MonadZip type class. I also proposed several laws – some of them inspired by category theory and some by my work on F# joinads – hoping that this article may contribute to the discussion about the laws required by MonadZip.

Acknowledgements

Thanks to Alan Mycroft for inspiring discussion about some of the monads demonstrated in this article and to Dominic Orchard for many useful comments on a draft of the article as well as discussion about category theory and the `mzip` laws. I'm

also grateful to Brent Yorgey for proofreading the article and suggesting numerous improvements.

References

[1] Tomas Petricek and Don Syme. Joinads: A retargetable control-flow construct for reactive, parallel and concurrent programming. In **PADL'11**, pages 205–219 (2011).

[2] Bring back monad comprehensions. `http://tinyurl.com/ghc4370`.

[3] Nils Schweinsberg. Fun with monad comprehensions (2010). `http://blog.n-sch.de/2010/11/27/fun-with-monad-comprehensions/`.

[4] Simon L. Peyton Jones and Philip Wadler. Comprehensive comprehensions. In **Haskell'07**, pages 61–72 (2007).

[5] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. **Journal of Functional Programming**, 8(4):pages 437–444 (July 1998).

[6] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In **Advanced Functional Programming, Second International School-Tutorial Text**, pages 184–207 (1996).

[7] S. Doaitse Swierstra. Combinator parsing: A short tutorial. Technical report, Utrecht University (2008). `http://tinyurl.com/parsing-tutorial`.

[8] Conor Mcbride and Ross Paterson. Applicative programming with effects. **Journal of Functional Programming**, 18:pages 1–13 (2007).

[9] Wikipedia. Pumping lemma for context-free languages (2011). `http://en.wikipedia.org/wiki/Pumping_lemma_for_context-free_languages`.

[10] Koen Claessen. A poor man's concurrency monad. **Journal of Functional Programming**, 9:pages 313–323 (May 1999).

[11] William L. Harrison. Cheap (but functional) threads (2008). `http://www.cs.missouri.edu/~harrison/drafts/CheapThreads.pdf`. Submitted for publication.

[12] Nikolaos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In **3rd Panhellenic Logic Symposium** (2001).

[13] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: better strategies for parallel haskell. In **Haskell'10**, pages 91–102. ACM (2010).

[14] `http://hackage.haskell.org/package/parallel`.

[15] Wikipedia. Monoidal functor (2011). `http://en.wikipedia.org/wiki/Monoidal_functor`.

[16] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In **ICFP'09**, pages 35–46 (2009).