Extending Monads with Pattern Matching

Tomas Petricek   Alan Mycroft
University of Cambridge
{tomas.petricek, am}@cl.cam.ac.uk

Don Syme
Microsoft Research Cambridge
don.syme@microsoft.com

Abstract

Sequencing of effectful computations can be neatly captured using monads and elegantly written using **do** notation. In practice such monads often allow additional ways of composing computations, which have to be written explicitly using combinators.

We identify joinads, an abstract notion of computation that is stronger than monads and captures many such ad-hoc extensions. In particular, joinads are monads with three additional operations: one of type $m\ a \rightarrow m\ b \rightarrow m\ (a, b)$ captures various forms of *parallel composition*, one of type $m\ a \rightarrow m\ a \rightarrow m\ a$ that is inspired by *choice* and one of type $m\ a \rightarrow m\ (m\ a)$ that captures *aliasing* of computations. Algebraically, the first two operations form a near-semiring with commutative multiplication.

We introduce **docase** notation that can be viewed as a monadic version of **case**. Joinad laws imply various syntactic equivalences of programs written using **docase** that are analogous to equivalences about **case**. Examples of joinads that benefit from the notation include speculative parallelism, waiting for a combination of user interface events, but also encoding of validation rules using the intersection of parsers.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Control structures; F.1.2 [*Models of Computation*]: Parallelism and concurrency

General Terms Languages, Theory

1. Introduction

Monads are traditionally used for embedding sequential computations into lazy functional code, but many recent uses go well beyond sequencing of state or computations. Monads have been used for the exact opposite—to explicitly specify parallelism. This is done by taking a core sequential monad and adding combinators that increase the expressive power beyond sequencing.

Monads for concurrent [5] and parallel programming [15] support forking and synchronizing computations [4]. A monad for user-interface programming includes combinators for merging events from various sources [29]. These ad-hoc extensions are extremely useful, but they are not uniform. Developers have to understand different combinators for every computation and they lose the syntactic support provided by **do** notation.

This paper discusses *joinads*—an abstract notion of computations that extends monads. Joinads capture a pattern that appears in many monadic computations with an additional expressive power that goes beyond sequencing. We presented an earlier form of joinads in F$^{\#}$[27]. This paper makes several novel findings, but our first contribution to Haskell is similar to the earlier work in F$^{\#}$:

- We add language support for important kinds of computations, including parallel, concurrent and reactive programming. This is done via a lightweight, reusable language extension that builds on core functional concepts such as pattern matching.

This paper simplifies the concept of joinad and requires that every joinad is also a monad (just like every group is also a monoid). In Haskell, we also relate several ideas that already disconnectedly exist. The specific new contributions of this paper are:

- We present **docase** notation for Haskell[1] (Sections 2, 4) that allows programming with monadic computations extended with aliasing, parallel composition and choice. We specify laws about these operations to guarantee that **docase** keeps the familiar semantics of pattern matching using **case** (Section 5).
- To demonstrate the usefulness of the extension, we consider parsing (Section 3.1), GUI programming using events (Section 3.2), lightweight concurrency (Section 3.4), and a parallelism monad with support for speculative parallelism (Section 3.3).
- The type of the above computations is captured by a Joinad type class (Section 4.2). It relates type classes that have been already proposed for Haskell. Based on our experience, we propose and discuss several adjustments to the Haskell base library and laws required by the type classes we combine (Section 8).
- A joinad is an abstract computation that extends monads with three operations. Deriving the laws about the three operations (Section 6) reveals that two of the operations form an algebraic structure known as a *near-semiring*.

The following section demonstrates the usefulness of **docase** in the context of parallel programming.

2. Motivating example

Consider the following problem: we are given a tree with values in leaves and we want to test whether a predicate holds for all values in the tree. This can be implemented as a recursive function:

$$\begin{aligned}
&\mathit{all} :: (a \rightarrow \mathit{Bool}) \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Bool}\\
&\mathit{all}\ p\ (\mathit{Leaf}\ v) = p\ v\\
&\mathit{all}\ p\ (\mathit{Node}\ \mathit{left}\ \mathit{right}) = \mathit{all}\ p\ \mathit{left} \wedge \mathit{all}\ p\ \mathit{right}
\end{aligned}$$

The execution of the two recursive calls could proceed in parallel. Moreover, when one of the branches completes returning *False*, it is not necessary to wait for the completion of the other branch as the overall result must be *False*.

Haskell'11, September 22, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0860-1/11/09...$10.00

[1] Prototype version is available at `http://github.com/tpetricek/Haskell.Joinads` and we plan to submit a GHC patch in the future.

Running two branches in parallel can be specified easily using strategies [21, 32], but adding short-circuiting behaviour is challenging. Using the **docase** notation and a monad for parallel programming, the problem can be solved as follows:

```
all :: (a → Bool) → Tree a → Par Bool
all p (Leaf v)          = return (p v)
all p (Node left right) =
  docase (all p left, all p right) of
    (False, ?)   → return False
    (?, False)   → return False
    (allL, allR) → return (allL ∧ allR)
```

The function builds a computation annotated with hints that specify how to evaluate it in parallel using the *Par* monad [15] extended with the support for non-deterministic choice operator [26].

To process sub-trees in parallel, the snippet constructs two computations (of type *Par Bool*) and uses them as arguments of **docase**. Patterns in the alternatives correspond to individual computations. A special pattern ? denotes that a value of the monadic computation does not have to be available for the alternative to be selected. When the processing of the left subtree completes and returns *False*, the first alternative can be selected immediately, because the result of the second computation is not required.

If the result of the left subtree is *True* and the right one has not completed, none of the alternatives are immeriately enabled. After the right subtree is processed, one of the last two alternatives can be selected. The choice operator added to the *Par* monad is non-deterministic, so the programmer needs to provide alternative clauses that produce the same result in case of race. We return to this topic in Section 3.3, but it is, the case in the above example.

The selection between alternative clauses is done using the *choice* operator. Note that the result of each computation is used in two independent alternatives. Evaluating the argument repeatedly would defeat the purpose of **docase**, so the translation uses the *aliasing* operator to avoid this. The third alternative combines two computations, which is achieved using *parallel composition* operator provided by the *Par* monad.

The translation of **docase** is more complex than of the **do** notation. This is not a bad thing—the notation can be used to write programs that would otherwise be very complex. In the above example, developers would typically write the solution in a more imperative style shown in Appendix A. The length of the explicit version is 21 lines compared to 6 line in the version above.

3. Introducing docase

This section introduces **docase** using four examples. We first consider **docase** expressions with a single alternative that can be also written using *zip comprehensions* [9] and then gradually add remaining features. A formal definition is shown in Section 4.

3.1 Parallel composition of parsers

Parsers are a common example of monads. A parser is a function: when supplied with an input, it returns a parsed value and the remaining unconsumed input. The following definition largely follows the one by Hutton and Meijer [30]:

```
newtype Parser a = P (String → [(a, Int, String)])
```

Compared to standard parsers, there is one notable difference. In addition to the parsed result and unconsumed input, the result also contains *Int* value, which denotes the number of consumed characters. This will be needed later. A parser can be made an instance of Monad to allow sequential composition and an instance of MonadPlus to support choice. A more interesting question is, what does a parallel composition of parsers mean:

```
instance MonadZip Parser where
  mzip (P p1) (P p2) = P (λinp →
    [((a, b), num1, tail1) |
      (a, num1, tail1) ← p1 inp,
      (b, num2, tail2) ← p2 inp, num1 ≡ num2])
```

Figure 1. Instance of MonadZip for parsers

```
mzip :: Parser a → Parser b → Parser (a, b)
```

Judging by the type, the *mzip* operation could be implemented in terms of ≫= and *return*. This implementation would not, in general, obey the laws we require. We give more details in Section 6.1. For parsers, the *mzip* operation parses the input using both parsers and then returns all combination of values such that the two parsers consumed the same number of input characters. The meaning of this operation is that it creates a parser for a language that is an *intersection* of languages described by the two parsers.

This implementation of *mzip* for parsers is shown in Figure 1. It applies the two parsing functions to the same input and then returns all combinations for which the predicate *num1* ≡ *num2* holds. An alternative implementation could compare the tails, but that would be inefficient and would not work for infinite inputs.

The function belongs to the MonadZip type class that has been added to GHC as part of a recent implementation of *monad comprehensions*. Monad comprehensions [36] generalize list comprehensions to work with an arbitrary monad. The recent extension [1, 9] also generalizes grouping and ordering [16] and syntax for zipping (*zip comprehensions*), hence the name *mzip*. To demonstrate the parallel between **docase** and generalized monad comprehensions, we start with an example written using both notations.

Example. Cambridge telephone numbers can be specified as strings satisfying three independent rules: they consist of 10 characters, contain only digits and they start with the prefix 1223. The following snippet shows how to encode this rule using both *parallel monad comprehensions* and the **docase** notation:

```
valid = docase (many (sat isDigit),
                replicateM 10 item,
                startsWith (string "1223"))
  of (num, _, _) → return num
valid = [num | num ← many (sat isDigit)
             | _   ← replicateM 10 item
             | _   ← startsWith (string "1223")]
```

The three arguments of the **docase** construct are combined using the *mzip* function. In zip comprehensions, the same role is played by the bar symbol. If the parsers succeed, they return the same string, so the snippet only needs the result of a single parser. The **docase** snippet ignores other values using _ patterns instead of ? patterns. The ? pattern is special and it specifies that a value is not required, which means that the parser can fail. Conversely, the _ pattern requires the parser to succeed, but then ignores the value.

The **docase** notation makes it possible to write everything that can be written using zip comprehensions in a style similar to **do** notation, but it also adds additional expressive power in a different way than generalized monad comprehensions.

Desugaring. The desugaring of **docase** in the simple case shown above is essentially the same as desugaring of parallel monad comprehensions. In the translation, the *mzip* operation is written as ⊗. The reason for this will become clear when we discuss the algebraic theory behind joinads in Section 7.3.

$$\begin{aligned}&\mathit{validPhone} = \\ &\quad ((\mathit{many}\ (\mathit{sat}\ \mathit{isDigit}) \otimes \mathit{times}\ \mathit{item}\ 10) \\ &\qquad \otimes \mathit{startsWith}\ (\mathit{string}\ \texttt{"1223"})) \gg\!\!= \lambda x \to \\ &\qquad\quad \mathbf{case}\ x\ \mathbf{of}\ ((\mathit{num}, \_), \_) \to \mathit{return}\ \mathit{num}\end{aligned}$$

The actual translation includes several additional features that are explained later, but they have no effect on the meaning. The expression combines all arguments of **docase** (or all parallel generators of a comprehension) using the $\otimes$ operation. The result is a combined value of type $\mathit{Parser}\ ((\mathit{String}, \mathit{String}), \mathit{String})$. This value is passed as an input to $\gg\!\!=$. The lambda function decomposes the tuple using original patterns of the **docase** alternative. In this example, the pattern never fails, so other cases are omitted. The body of the lambda creates a parser that succeeds and returns the parsed valid phone number.

Next, consider a case where **docase** has multiple alternatives, but each contains only a single binding (a pattern other than ?).

3.2 Choosing between events

The examples in this section are based on the imperative stream monad developed by Scholz [29]. Imperative streams are "a generalization of the IO monad suitable for synchronous concurrent programming". An imperative stream produces zero or more values and performs side-effects at certain (discrete) times. Our example uses a simplified model of *event streams* with type $\mathit{Evt}\ a$ that do not allow side-effects.

Event streams can be viewed as functions that take a time indicating when they are started and return a list of time value pairs representing the occurrences of the event. They are instances of the Monad type class. The return operation creates an event stream that occurs exactly once at the time when it was started. The behaviour of the $\gg\!\!=$ operation is as follows: when the input event occurs, the event stream returned by $\gg\!\!=$ starts producing the occurrences of the event stream generated by the function passed to $\gg\!\!=$ until the next occurrence of the input event.

In addition to the operations of Monad, event streams can also implement a monadic or–else operation representing a choice:

$$\mathit{morelse} :: \mathit{Evt}\ a \to \mathit{Evt}\ a \to \mathit{Evt}\ a$$

The resulting event stream occurs whenever any of the two arguments occur. When both of the arguments occur at the same time, then the returned value is the value produced by the first (left) argument. As explained later (Section 5), this natural left bias of the operation is required by a law about $\mathit{morelse}$.

Example. Assume that the user can create objects by clicking and can use the Shift key to switch between two types of objects. The user interface provides event streams $\mathit{shiftDown}$ and $\mathit{shiftUp}$ that occur when Shift is pressed and released; an event stream load occurs once when the application starts and $\mathit{mouseClick}$ occurs each time the mouse button is pressed.

The following snippet creates an event stream of type $\mathit{Evt}\ \mathit{Bool}$ that occurs each time a mouse button is clicked. The value carried by the event is a flag denoting whether Shift was pressed:

$$\begin{aligned}&\mathit{shiftClicks} = \mathbf{docase}\ (\mathit{load}, \mathit{shiftUp}, \mathit{shiftDown})\ \mathbf{of} \\ &\quad (a, ?, ?) \to \mathit{fmap}\ (\mathit{const}\ \mathit{False})\ \mathit{mouseClick} \\ &\quad (?, u, ?) \to \mathit{fmap}\ (\mathit{const}\ \mathit{False})\ \mathit{mouseClick} \\ &\quad (?, ?, d) \to \mathit{fmap}\ (\mathit{const}\ \mathit{True})\ \mathit{mouseClick}\end{aligned}$$

When one of the events passed to **docase** produces a value, the resulting event starts producing values generated by one of the alternatives (True or False whenever mouse is clicked). Each of the alternatives matches on a single event stream and ignores the values of other event streams using the ? pattern. The variables bound by the patterns are not used, so we could use _, but naming the variables makes the example easier to follow.

$$\begin{aligned}&[\![\mathit{unit}\ v]\!]_s = \lambda t \to \mathbf{if}\ s \equiv t\ \mathbf{then}\ \mathit{Just}\ v\ \mathbf{else}\ \mathit{Nothing} \\ &[\![a \otimes b]\!]_s = \lambda t \to \mathbf{case}\ ([\![a]\!]_s\ t, [\![b]\!]_s\ t)\ \mathbf{of} \\ &\quad (\mathit{Just}\ v_1, \mathit{Just}\ v_2) \to \mathit{Just}\ (v_1, v_2);\ \ \_ \to \mathit{Nothing} \\ &[\![a \oplus b]\!]_s = \lambda t \to \mathbf{case}\ ([\![a]\!]_s\ t, [\![b]\!]_s\ t)\ \mathbf{of} \\ &\quad (\mathit{Just}\ v_1, \_) \to \mathit{Just}\ v_1;\ \ (\_, o_2) \to o_2 \\ &[\![a \gg\!\!= f]\!]_s = \lambda t \to \mathbf{case}\ (\mathit{last}\ t\ [\![a]\!]_s)\ \mathbf{of} \\ &\quad (\mathit{Just}\ (t_1, v_1)) \to [\![f\ v_1]\!]_{t_1}\ t;\ \ \_ \to \mathit{Nothing} \\ &\ \ \mathbf{where}\ \mathit{last}\ 0\ \_ = \mathit{Nothing} \\ &\qquad\quad \mathit{last}\ t\ \mathit{sf} = \mathbf{case}\ \mathit{sf}\ t\ \mathbf{of}\ \mathit{Just}\ v \to \mathit{Just}\ (t, v) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \_ \to \mathit{last}\ (t-1)\ \mathit{sf}\end{aligned}$$

Figure 2. Semantics of imperative streams

Desugaring. The desugared code is shown below. Each alternative binds only on a single event, so the translation does not use the mzip operation. The $\mathit{morelse}$ operation is abbreviated as $\oplus$:

$$\begin{aligned}&\mathit{shiftClicks} = \\ &\quad (\mathit{load} \gg\!\!= \lambda a \to \mathit{fmap}\ (\mathit{const}\ \mathit{False})\ \mathit{mouseClick}) \oplus \\ &\quad (\mathit{shiftUp} \gg\!\!= \lambda u \to \mathit{fmap}\ (\mathit{const}\ \mathit{False})\ \mathit{mouseClick}) \oplus \\ &\quad (\mathit{shiftDown} \gg\!\!= \lambda d \to \mathit{fmap}\ (\mathit{const}\ \mathit{True})\ \mathit{mouseClick})\end{aligned}$$

The translator processes alternatives independently and then merges them using $\oplus$. The event stream corresponding to a binding pattern (pattern other than ?) is passed as the first argument to $\gg\!\!=$. The provided function contains the body of the alternative. The example is simplified, because patterns in the alternatives do not fail. If pattern matching could fail, the event stream should continue behaving according to the last selected alternative. To encode this behaviour, the translation needs one more extension (Section 3.4).

Semantics. Showing a complete implementation of event streams is beyond the scope of this article. We present a semantics that defines the implementation and can be used to verify that the operations obey joinad laws. The semantics follows the original definition of imperative streams [29]. Instead of using lists, we model event occurrences as a function returning Maybe value:

$$[\![\mathit{Evt}\ a]\!]_T :: T \to \mathit{Maybe}\ a$$

The time T is a discrete value. When applied to a starting time $t \in T$, the semantic function gives a partial function that returns $\mathit{Just}\ v$ if the event occurs at the specified time. The semantics of Monad operations, $\otimes$ and also $\oplus$ is given in Figure 2. The semantics of $\otimes$ and $\oplus$ follow a similar pattern. At given time, they combine both, or take the leftmost value if the required values are available. Finally, the result of monadic bind ($\gg\!\!=$) behaves as an event stream generated by the last occurrence of the input event.

Using this semantic model, we could derive an implementation using the techniques developed recently for functional reactive programming (FRP) by Elliott [6]. Compared to other approaches, imperative streams give a simple model based just on discrete events, but the **docase** notation can be also used when programming with continuous values.

3.3 Aliasing parallel computations

This section explains the use of the last of the three joinad operations: malias which represents aliasing of computations. The operation gives the monad (joinad) more control of the control-flow by abstracting away certain aspect of the evaluation mechanism. The parallel all function in Section 2 critically relied on this feature, so we demonstrate the problem using the parallelism monad.

A value of type $\mathit{Par}\ a$ represents a computation that can be evaluated (using some parallel evaluator) to get a value of type a.

$$
\begin{aligned}
&[\![unit\ v]\!] &&= \lambda t \to (t, v)\\
&[\![mzero]\!] &&= \lambda t \to (\infty, \bot)\\
&[\![a \otimes b]\!] &&= \lambda t \to (max\ t_1\ t_2, (v_1, v_2))\\
&\quad \mathbf{where}\ ((t_1, v_1), (t_2, v_2)) = ([\![a]\!]\ t, [\![b]\!]\ t)\\
&[\![a \oplus b]\!] &&= \lambda t \to (min\ t_1\ t_2, v)\\
&\quad \mathbf{where}\ ((t_1, v_1), (t_2, v_2)) = ([\![a]\!]\ t, [\![b]\!]\ t)\\
&\qquad\qquad v \mid t_1 \leqslant t_2 = v_1 \mid otherwise = v_2\\
&[\![a \gg\!= f]\!] &&= \lambda t \to [\![b]\!]\ s\\
&\quad \mathbf{where}\ (s, v) = [\![a]\!]\ t;\quad b = f\ v\\
&[\![malias\ a]\!] &&= \lambda t \to (\lambda t_2 \to (max\ t_1\ t_2, v))\\
&\quad \mathbf{where}\ (t_1, v) = [\![a]\!]\ t
\end{aligned}
$$

Figure 3. Semantics of futures

Parallel computations are instances of Monad. The *return* operation creates a computation that immediately returns and $\gg\!=$ evaluates the argument and then evaluates the result produced by a continuation. The implementation of *mzip* for $Par\ a$ starts two computations in parallel and produces a value when they both complete; *morelse* represents a non-deterministic choice and completes when the first of the two computations produce a value.

Example. Consider some calculation, that uses a main function, *calc*, and two alternative heuristic functions, *alt1* and *alt2*. In order to continue, we need the result of the main function and one heuristic. Using **docase**, this can be written as follows:

$$
\begin{aligned}
&calcAlt\ inp = \mathbf{docase}\ (calc\ inp, alt1\ inp, alt2\ inp)\ \mathbf{of}\\
&\quad (a, b, ?) \to return\ (a, b)\\
&\quad (a, ?, c) \to return\ (a, c)
\end{aligned}
$$

Note that the first argument is bound to a variable a in both of the alternatives. The desired operational meaning is that the expression starts all three computations in parallel and then waits until computations required by some alternative complete. Using the logic described so far, the snippet might be translated as follows:

$$
\begin{aligned}
&calcAlt\ inp =\\
&\quad (calc\ inp) \otimes (alt1\ inp) \gg\!= \lambda(a, b) \to return\ (a, b) \oplus\\
&\quad (calc\ inp) \otimes (alt2\ inp) \gg\!= \lambda(a, c) \to return\ (a, c)
\end{aligned}
$$

This does not give the required behaviour. The code creates a computation that starts four tasks – the two alternative heuristics and two instances of the main computation. Eliminating a common subexpression *calc inp* does not solve the problem. The value $Par\ a$ obtained from *calc inp* represents a recipe for creating a computation, as opposed to a running task. When used repeatedly, it starts a new computation.

Desugaring. To get the desired semantics, we need some way to start the computation once and get an aliased computation that can be used multiple times. This is exactly what the *malias* operation provides. It can be best explained by looking at the type signature together with the implementation for the $Par\ a$ monad:

$$
\begin{aligned}
&malias :: Par\ a \to Par\ (Par\ a)\\
&malias\ p = \mathbf{do}\\
&\quad v \leftarrow spawn\ p\\
&\quad return\ (get\ v)
\end{aligned}
$$

The implementation starts a given computation using the *spawn* function, which returns a mutable variable that will contain the result of the computation when it completes. Then it returns a computation of type $Par\ a$ created using the *get* function. When used, the computation blocks until the variable is set.

Equipped with this operation, the desugaring can create an aliased monadic computation for each of the **docase** arguments and then use the aliased computations repeatedly:

$$
\begin{aligned}
&calcAlt\ inp =\\
&\quad malias\ (calc\ inp) \gg\!= \lambda c0 \to\\
&\quad malias\ (alt1\ inp) \gg\!= \lambda c1 \to\\
&\quad malias\ (alt2\ inp) \gg\!= \lambda c2 \to\\
&\quad c0 \otimes c1 \gg\!= \lambda(a, b) \to return\ (a, b) \oplus\\
&\quad c0 \otimes c2 \gg\!= \lambda(a, b) \to return\ (a, b)
\end{aligned}
$$

This version gives the desired operational behaviour. Each of the three arguments of **docase** is started exactly once (in the implementation of *malias*). The body is composed using computations that merely represent aliases (using a mutable variable internally). In particular, both of the alternatives combined using $\oplus$ use the alias *c0* that refers to the *calc* computation.

Semantics. To describe the example more formally, we present a simple semantics. It can be used to verify that the joinad laws (Section 6) hold for the $Par\ a$ type. Here, a computation is modelled as a function that takes a time when the computation is started and returns a time when it completes together with the result:

$$[\![Par\ a]\!] :: T \to (T, a)$$

The semantics is shown in Figure 3. Operations of Monad as well as $\otimes$ and $\oplus$ behave as already informally described. The *malias* operation applies the semantic function to a given computation with the starting time of *malias* as an argument. The resulting computation finishes either at the completion time or at the time when it is created, whichever happens later.

The semantics does not capture the number of computations running in parallel, so it is only useful for considering joinad laws. The next section describes a variation where computations have side-effects. In that case *malias* becomes more important, because it avoids duplication of side-effects. For some monads, such as IO, the *malias* operation can be defined as follows:

$$malias\ op = op \gg\!= return \circ return$$

This definition could be used for any monad, but it would not always give useful behaviour. For example, in $Par\ a$, it would unnecessarily sequentialize all computations.

Nondeterminism. The semantics presented in Figure 3 is deterministic, but in reality, this is not the case. We could add small δ to all operations involving time. The interesting case is the $\oplus$ operation, where the value (second element of the tuple) depends on the time. This means that the operation introduces non-determinism.

To keep programs simple and deterministic, we follow the approach used by Elliott for the unambiguous choice (*unamb*) operator [6]. The *morelse* operation can be used only when the two arguments are *compatible*:

$$
\begin{aligned}
&compatible\ a\ b = \forall\ t\ .\ (t_1 \equiv \infty) \vee (t_2 \equiv \infty) \vee (v_1 \equiv v_2)\\
&\quad \mathbf{where}\ (t_1, v_1) = [\![a]\!]_t;\quad (t_2, v_2) = [\![b]\!]_t
\end{aligned}
$$

When started at the same arbitrary time, the two operations are required to produce the same value if they both complete. As discussed in the next section, an operation that never completes can be created using the *mzero* operation and represents an alternative clause with failing patterns. The condition could be reformulated in terms of **docase** alternatives. It is not difficult to see that the condition holds for the motivating example from Section 2.

3.4 Committing to a concurrent alternative

The final simplification made in the previous example was that the patterns of **docase** alternatives never failed. This aspect can be demonstrated using a monad based on Poor Man's Concurrency

Monad developed by Claessen [5]. A value of type $Concur\ m\ a$ represents a computation that will eventually produce a value of type a and may produce effects in a monad m along the way.

The monad is similar to parallel computations from the previous section. To give a concrete semantics, assume that the underlying monad is a writer monad using monoid M to keep the state. The semantics from the previous section could be extended by adding state to the result:

$$[\![Concur\ (Writer\ M)\ a]\!] :: T \to (T, a, M)$$

The semantics from Figure 3 can be extended in a straightforward way to use this function. Unlike $Par\ a$, the implementation of $Concur\ m\ a$ does not actually run computations in parallel. It emulates concurrency by interleaving of steps. This means that the $Concur\ m\ a$ monad is deterministic and the time T represents the number of primitive steps. In order to support patterns that may fail, the type also provides an operation $mzero$ representing a failure:

$$mzero :: Concur\ m\ a$$

The operation is defined for standard monads that implement the MonadPlus type class. For the Poor Man's Concurrency monad, the operation creates a computation that never produces a value.

Example Consider a function that concurrently downloads data from two servers. When some data is not available, the function returns *Nothing*. When both servers produce a value, the function returns *Just* containing a tuple, but only when the values are compatible. When data is incompatible, the function should fail.

```
downloadBoth :: Concur IO Bool
downloadBoth = docase (dl1, dl2) of
  (Just a, Just b) → do
    lift (print "Got both values")
    if compatible a b then return Just (a, b) else mzero
  (_, _) → do
    lift (print "Some value missing")
    return Nothing
```

The first alternative of **docase** matches when both of the computations produce *Just* value. The body prints a message (the *lift* function lifts a value $IO\ a$ into $Concur\ IO\ a$) and then returns *Just* or fails by returning *mzero*. The second alternative handles any values and returns *Nothing* after printing a message.

Tullsen [33] suggested returning *mzero* when pattern matching fails. When it succeeds, the original body is returned. For simplicity, we omit *malias*:

```
(dl1 ⊗ dl2 ≫= λt → case t of
  (Just a, Just b) → do ...
  _ → mzero) ⊕
(dl1 ⊗ dl2 ≫= λt → do ...)
```

An intuitive expectation about pattern matching that we want to keep for joinads is that → behaves as a *commit point*. Once arguments match patterns of some alternative, the code will execute this alternative and not any other. However, this is not the case with the desugaring above. When the function downloads two incompatible values, it prints "Got both values". Then it fails and starts executing the second clause, printing "Some values missing".

Desugaring To get the desired behaviour, the desugaring needs to add an additional level of wrapping. Instead of just returning the body, we wrap the body using *return*:

```
(dl1 ⊗ dl2 ≫= λt → case t of
  (Just a, Just b) → return (do ...)
  _ → mzero) ⊕
(dl1 ⊗ dl2 ≫= λt →
  return (do ...)) ≫= id
```

The cases where pattern matching succeeded now contain just a call to *return* with the body of the alternative as an argument and the cases where the pattern matching fails contain *mzero*. This means that the type of values aggregated using $\oplus$ is $m\ (m\ a)$. Additionally, all of them are either of the form $m \gg\!= return \circ f$ or $m \gg\!= mzero$. The result of $\oplus$ has the same type as its arguments, so the overall result also has a type $m\ (m\ a)$. It represents a monadic value that wraps (or produces) body(ies) that have been selected. To create a computation that actually runs the body, the desugaring inserts $\gg\!= id$ at the end of the translated expression.

4. Language extension

This section formally defines the **docase** notation including its syntax, typing rules and the translation.

4.1 Syntactic extension

The extension adds an additional syntactic case to Haskell expression e. It also defines a category of docase alternatives a and docase patterns w that include the additional special pattern ? (ignore):

$p\ = x \mid (p_1, ..., p_n) \mid ...$	Ordinary patterns
$w\ = \ ?$	Monadic ignore pattern
$\quad\mid\ p$	Monadic binding pattern
$a\ = (w_1, ..., w_n) \to e$	Docase alternative ($\exists i : w_i \neq ?$)
$e\ = \textbf{docase}\ (e_1, ..., e_n)\ \textbf{of}\ a_1; ...; a_k$	Docase expression with k alternatives ($k \geqslant 1$)

The **docase** expression is similar to standard **case**. A docase pattern w can be standard Haskell patterns p or a special *ignore pattern* written as ?. A docase alternative a must contain at least one binding pattern (pattern other than ?), because there is no easy way to construct monadic computation that succeeds when all other computations fail. Finally, the **docase** expression must include at least one alternative.

4.2 Joinad type class

The **docase** syntax operates on values of some type $m\ a$ that is an instance of a Joinad type class. The type class provides operations required by the **docase** translation. Figure 4 shows the definition of Joinad. The definition just combines several classes that already exist in various Haskell extensions and packages.

- MonadZero and MonadOr are defined in a MonadPlus reform proposal [11]. It aims to distinguish between cases when the (monoidal) operation is unbiased (MonadPlus) and when it has a left bias (MonadOr). For joinads, we require left bias, but we express the law slightly differently (Section 6.2).
- MonadZip is defined by a GHC extension that adds *monad comprehensions* [1, 9]. The extension adds new expressive power that is not available with the **do** notation [25]. The **docase** syntax uses the MonadZip type class in a similar way as *parallel monad comprehension* and provides similar expressivity using a syntax similar to the **do** notation.
- MonadAlias is similar to the Extend type class from the comonad package [18]. The only difference is that we require the type to also be a monad.

The theoretical background and the laws that are required to hold about the operations are discussed in Sections 5 and 7. The next two sections complete the specification of the language extension.

4.3 Typing rules

Similarly to other syntactic sugar in Haskell [16], the **docase** expression is type-checked before translation. The typing rules are shown in Figure 5 and are defined in terms of three judgements.

$$\begin{array}{l}
\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadZero}\ m \textbf{ where} \\
\quad \mathit{mzero} :: m\ a \\
\textbf{class } \mathit{MonadZero}\ m \Rightarrow \mathit{MonadOr}\ m \textbf{ where} \\
\quad \mathit{morelse} :: m\ a \to m\ a \to m\ a \\
\textbf{class } \mathit{Monad} \Rightarrow \mathit{MonadZip}\ m \textbf{ where} \\
\quad \mathit{mzip} :: m\ a \to m\ b \to m\ (a, b) \\
\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadAlias}\ m \textbf{ where} \\
\quad \mathit{malias} :: m\ a \to m\ (m\ a) \\
\textbf{class } (\mathit{MonadAlias}\ m, \mathit{MonadZip}\ m, \mathit{MonadOr}\ m) \\
\quad \Rightarrow \mathit{Joinad}\ m
\end{array}$$

Figure 4. The definition of Joinad type class.

The judgement $\vdash w : \tau \Rightarrow \Delta$ for patterns is similar to the one used by Wadler and Peyton Jones [16]. It specifies that a pattern w of type τ binds variables of the environment Δ. An ignore pattern does not bind any variables (IGN); a variable pattern binds a single variable (VAR) and a tuple pattern binds the union of variables bound by sub-patterns (TUP).

The judgement $\Gamma, m, \bar{\tau} \vdash a \rightsquigarrow \alpha$ is more interesting. It checks the type of an individual alternative of the **docase** construct. The judgement is provided with an environment Δ, a Joinad type m and a list of types of **docase** arguments. It type-checks the alternative and yields the type of values produced by the body of the alternative (ALT). The body e of each alternative must have the same monadic type m (of kind $* \to *$) as the **docase** arguments.

Finally, $\Gamma \vdash a : \tau$ extends the standard type-checking procedure for Haskell expressions with a rule for **docase** (DOC). When the type of arguments is a Joinad type m (of kind $* \to *$) applied to some type argument and all alternatives yield the same return type α, then the overall type of the expression is $m\ \alpha$.

4.4 Translation

After type-checking, the **docase** notation is translated to applications of functions provided by the Joinad type class. The desugaring is defined using two functions:

$$\begin{array}{l}
d\langle - \rangle :: e \to e \\
c\langle - \rangle :: a \to [id] \to e
\end{array}$$

The first function takes an expression. If the argument is the **docase** expression, the function produces an expression that does not contain **docase** at the top-level. The second function is used for translating alternatives of **docase**. It takes a list of identifiers that refer to the arguments of the **docase** expression. The translation is defined by the following two rules:

$$\begin{array}{l}
d\langle \textbf{docase}\ (e_1, ..., e_n)\ \textbf{of}\ a_1; ...; a_k \rangle = \\
\quad \mathit{malias}\ e_1 \gg\!= \lambda v_1 \to\ ...\ \ \mathit{malias}\ e_n \gg\!= \lambda v_n \to \\
\quad (c\langle a_1 \rangle\ [v_1, ..., v_n]\ \oplus ... \oplus\ c\langle a_n \rangle\ [v_1, ..., v_n]) \gg\!= \mathit{id}
\end{array}$$

$$\begin{array}{l}
c\langle (w_1, ..., w_n) \to e \rangle\ [v_1, ..., v_n] = \\
\quad v_1\ \otimes ... \otimes\ v_m \gg\!= \lambda x \to \textbf{case}\ x\ \textbf{of} \\
\qquad (p_1, ..., p_m) \to \mathit{return}\ e \\
\qquad \mathit{otherwise} \quad \to \mathit{mzero} \\
\quad \textbf{where}\ [(p_1, v_1), ..., (p_m, v_m)] = \\
\qquad\qquad [(w_i, v_i) \mid i \leftarrow 1 ... n, w_i\ \neq ?]
\end{array}$$

The arguments $(e_1, .., e_n)$ of **docase** are first passed to malias, which constructs a value of type $m\ (m\ a)$. The $\gg\!=$ operator provides the lambda with values v_i that represents the aliased computations. The function $c\langle - \rangle$ takes an alternative and the aliased computations and produces values of type $m\ \ (m\ \ a)$ that represent monadic values carrying bodies to be executed. The results are combined using the $\oplus$ operation, which gives a value of type $m\ \ (m\ \ a)$. The last binding passes it to the identity function to execute the body of the selected alternative.

$$\boxed{\vdash w : \tau \Rightarrow \Delta}$$

$$\frac{}{\vdash ? : \tau \Rightarrow \{\}}\ (\text{IGN}) \qquad \frac{}{\vdash x : \tau \Rightarrow \{x : \tau\}}\ (\text{VAR})$$

$$\frac{\vdash w_i : \tau_i \Rightarrow \Delta_i}{\vdash (w_1, \ldots, w_n) : (\tau_1, \ldots, \tau_n) \Rightarrow \Delta_1 \cup \ldots \cup \Delta_n}\ (\text{TUP})$$

$$\boxed{\Gamma, m, \bar{\tau} \vdash a \rightsquigarrow \alpha}$$

$$\frac{\vdash w_i : \tau_i \Rightarrow \Delta_i \qquad \Delta_1 \cup \ldots \cup \Delta_n \vdash e : m\ \alpha}{\Gamma, m, \bar{\tau} \vdash (w_1, \ldots, w_n) \to e \rightsquigarrow \alpha}\ (\text{ALT})$$

$$\boxed{\Gamma \vdash a : \tau}$$

$$\frac{\langle \textbf{Joinad}\ m \rangle \qquad \Gamma \vdash e_i : m\ \tau_i \qquad \Gamma, m, \bar{\tau} \vdash a_i \rightsquigarrow \alpha}{\Gamma \vdash \textbf{docase}\ \bar{e}\ \textbf{of}\ a_1; \ldots; a_n : m\ \alpha}\ (\text{JON})$$

Figure 5. Typing rules for docase.

To translate an alternative, we identify which of the arguments are matched against a binding pattern. These computations are combined using the $\otimes$ operation. The resulting computation produces tuples such as $(a, (b, c))$. As discussed later, the $\otimes$ operation is associative, so the order of applying $\otimes$ does not matter. Values produced by the combined monadic computation are matched against a pattern re-constructed from *binding patterns* of the alternative. When a value matches, the body is wrapped using *return*. Otherwise, the alternative reports a failure using *mzero*.

5. Reasoning about monadic pattern matching

The **docase** syntax intentionally resembles **case** syntax and we would like to guarantee that the operational behaviour is similar as well. The notation is used for working with values of an abstract type, so there is no concrete semantics.

Figure 6 shows syntactic transformations that must preserve the semantics. In Section 6, we find a set of laws that implies the equivalences required here. Using a mathematical model, we proved that the equivalences follow from the primitive laws using the Coq theorem prover[2]. Finding a set of equivalences that permit proving the opposite implication (completeness) similarly to monad comprehensions [36] is left to future work.

- *Binding equivalence* describes a degenerate case in which pattern matching uses a single alternative that always succeeds. It specifies that a specific use of *malias* does not affect the meaning of monadic binding.
- *Argument ordering* specifies that the order in which arguments and patterns are specified does not affect the meaning. This equation implies commutativity and associativity laws of the $\otimes$ operation.
- Unlike the order of arguments, the order of clauses is important. The *clause ordering* equivalence specifies that the $\oplus$ operation is left-biased.

[2] `http://www.cl.cam.ac.uk/~tp322/papers/docase.html`

- The equivalences *alternative noninterference* and *argument noninterference* specify that including additional failing clause or argument, respectively, has no effect on the meaning. (In equation (4), the symbol $\bullet$ stands for a pattern that never succeeds.) The equations are manifested as laws that identify $mzero$ as *zero element* of $\otimes$ and *neutral element* of $\oplus$.
- The next three equivalences describe the case when arguments are created in some special way[3]. They define a group of *naturality* properties of the $\otimes$ operation.
- The *distributivity* equivalence requires that certain nested uses of **docase** can be flattened. This equivalence specifies that $\otimes$ distributes over $\oplus$.

6. Joinad laws

This section discusses primitive laws about individual joinad operations that are implied by the above equivalences. First, we review the well-known monad laws that are also required for any joinad:

$$unit\ a \gg\!= f \equiv f\ a \qquad \text{(left identity)}$$

$$m \gg\!= unit \equiv m \qquad \text{(right identity)}$$

$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= \lambda x \to f\ x \gg\!= g \qquad \text{(associativity)}$$

Joinad also requires the $mzero$ operation from MonadZero. The value should behave as a zero element with respect to binding:

$$mzero \gg\!= f \equiv mzero \qquad \text{(left zero)}$$

$$m \gg\!= \lambda x \to mzero \equiv mzero \qquad \text{(right zero)}$$

The *left zero* law is generally accepted. The *right zero* law is sometimes omitted, because it may not hold when m is $\bot$, but the official documentation for MonadPlus [2] includes it. All of these five laws are necessary to prove the equivalences in Figure 6.

6.1 MonadZip type class

This section discusses laws that are required about $\otimes$ by joinads. The laws presented here are also a reasonable requirement for monad comprehensions as the two are closely related. We give more details in Section 8.1, but many of the equivalences in Figure 6 can be rewritten using the *zip comprehension* syntax.

The first two laws allow arbitrary rearrangement of arguments aggregated using the $\otimes$ operation. This is required by *argument ordering* (2). The laws are expressed using two helper functions:

$$a \otimes (b \otimes c) \equiv map\ assoc\ ((a \otimes b) \otimes c) \qquad \text{(associativity)}$$

$$a \otimes b \equiv map\ swap\ (b \otimes a) \qquad \text{(symmetry)}$$

$$\textbf{where}\ assoc((a,b),c) = (a,(b,c))$$
$$swap(a,b) = (b,a)$$

As discussed in Section 7.1, these two laws are founded in mathematical theory behind joinads. The $\otimes$ operation is component of a *symmetric monoidal functor* that requires both of the above laws. Another law that is also required by this formalism is *naturality*, which is one of three laws that relate $\otimes$ to other operations:

$$map\ f\ a \otimes map\ g\ b \equiv map\ (f \times g)\ (a \otimes b) \qquad \text{(naturality)}$$

$$return\ a \otimes return\ b \equiv return\ (a,b) \qquad \text{(product)}$$

$$a \otimes a \equiv map\ dup\ a \qquad \text{(duplication)}$$

$$\textbf{where}\ dup\ a = (a,a)$$

These laws follow from the three *matching* equivalences and specify the result of applying $\otimes$ to specific monadic values:

[3] The map operation can be defined as $\gg\!= return$ and is also called $liftM$.

(1) Binding equivalence
$$\textbf{docase}\ m\ \textbf{of}\ v \to e \ \equiv\ \textbf{do}\ v \leftarrow m; e$$

(2) Argument ordering
$$\begin{array}{l}\textbf{docase}\ (m_1, ..., m_n)\ \textbf{of}\\ \quad (w_{1,\pi 1}\ ...\ w_{1,\pi n}) \to e_1; ...\\ \quad (w_{k,\pi 1}\ ...\ w_{k,\pi n}) \to e_k\end{array}$$
(are equivalent for any permutation π of $1 \ldots n$)

(3) Clause ordering
$$\begin{array}{rl} & \textbf{docase}\ m\ \textbf{of}\ v \to e_1;\ \ v \to e_2\\ \equiv & \textbf{docase}\ m\ \textbf{of}\ v \to e_1\end{array}$$

(4) Alternative noninterference
$$\begin{array}{rl} & \textbf{docase}\ m\ \textbf{of}\ v\ \to e_1;\ \ \bullet \to e_2\\ \equiv & \textbf{docase}\ m\ \textbf{of}\ \bullet \to e_2;\ \ v \to e_1\\ \equiv & \textbf{docase}\ m\ \textbf{of}\ v\ \to e_1\end{array}$$

(5) Argument noninterference
$$\begin{array}{rl} & \textbf{docase}\ (m, mzero)\ \textbf{of}\ (v, ?) \ \to e_1;\ \ (v_1, v_2) \to e_2\\ \equiv & \textbf{docase}\ (mzero, m)\ \textbf{of}\ (v_1, v_2) \to e_2;\ \ (?, v)\ \to e_1\\ \equiv & \textbf{docase}\ m\ \textbf{of}\ v \to e_1\end{array}$$

(6) Matching units
$$\begin{array}{rl} & \textbf{docase}\ (return\ e_1, return\ e_2)\ \textbf{of}\ (v_1, v_2) \to e\\ \equiv & \textbf{case}\ (e_1, e_2)\ \textbf{of}\ (v_1, v_2) \to e\end{array}$$

(7) Matching images
$$\begin{array}{rl} & \textbf{docase}\ (map\ f\ e_1, map\ g\ e_2)\ \textbf{of}\ (v_1, v_2) \to e\\ \equiv & \textbf{docase}\ (e_1, e_2)\ \textbf{of}\ (u_1, u_2) \to e\ [v_1 \leftarrow f\ u_1, v_2 \leftarrow g\ u_2]\end{array}$$

(8) Matching duplicate
$$\begin{array}{rl} & \textbf{docase}\ (a, a)\ \textbf{of}\ (u, v) \to e\\ \equiv & \textbf{docase}\ a\ \textbf{of}\ u \to e\ [v \leftarrow u]\end{array}$$

(9) Distributivity
$$\begin{array}{rl} & \textbf{docase}\ (m, n_1, n_2)\ \textbf{of}\\ & \quad (v, v_1, ?) \to e_1;\ \ v, ?, v_2 \to e_2\\ \equiv & \textbf{docase}\ (m, \textbf{docase}\ (n_1, n_2)\ \textbf{of}\\ & \quad (v_1, ?) \to return\ (\lambda v \to e_1);\\ & \quad (?, v_2) \to return\ (\lambda v \to e_2))\ \textbf{of}\ (v, f) \to (f\ v)\end{array}$$

Figure 6. Syntactic transformations that preserve the semantics.

- In *naturality*, the arguments are created using map. The application of $\otimes$ is lifted to be performed before the mapping. The mapping then transform both elements of the combined tuple.
- In *product*, the arguments are created using $return$. The combination of values is lifted to be performed before the use of $return$ and becomes a tuple constructor.
- In *duplication*, the arguments are two copies of the same monadic value. The duplication can be lifted inside the monad and performed on the actual values using map.

The next law specifies that $mzero$ is the *zero element* with respect to $\otimes$ (thanks to the symmetry of $\otimes$, it is both left and right zero):

$$a \otimes mzero \equiv mzero \equiv mzero \otimes a \qquad \text{(zero)}$$

This law is intuitively necessary. An $mzero$ value of type $m\ a$ does not contain any value of type a. Thus, given a value of type b, there is no way to construct a value of type (a, b).

Applicative functors. Given a monad, it is possible to define one instance of applicative functor (Applicative type class). An equivalent definition of this type class described by McBride and Paterson [22] defines an operation $\star$ that has exactly the same type as $\otimes$, but Applicative does not require *symmetry*.

The laws are different, hence the $\otimes$ operation is an independent addition to a monad. As discussed in an earlier paper [27], for commutative monads, the $\otimes$ operation can be defined using $\gg\!=$ and $return$, but this is not possible in general. For some types, the $\otimes$ operation is a part of a distinct Applicative instance. For example, $\star$ for a standard List monad is a Cartesian product, but $\otimes$ for lists is zip. This operation defines a distinct applicative functor (over the same type) named ZipList.

6.2 MonadOr type class

The MonadOr type class defines the $morelse$ operation (written as $\oplus$). As already discussed in Section 4.2, it represents a left-biased monoid. The monoid operation should be associative and have a unit element ($mzero$ in case of joinads). In Haskell, it should also obey a form of naturality law:

$$(u \oplus v) \oplus w \equiv u \oplus (v \oplus w) \qquad \text{(associativity)}$$
$$(map\ f\ u) \oplus (map\ f\ v) \equiv map\ f\ (u \oplus v) \qquad \text{(naturality)}$$
$$u \oplus mzero \equiv\ u \equiv mzero \oplus u \qquad \text{(unit)}$$

The *unit* law is required by the *alternative noninterference* (4) equivalence; the *naturality* is needed by multiple equivalences including *distributivity* (9). Finally, the *associativity* law does not directly correspond to any equivalence, but it specifies that the bracketing does not matter when aggregating the alternatives using $\oplus$ and makes this an unimportant implementation detail.

The left bias of the $\oplus$ operation is required by *clause ordering* (3). The equivalence gives the following law:

$$u \oplus map\ f\ u \equiv u \qquad \text{(left bias)}$$

The law considers a monadic value and an image created using map. When choosing between the two, the $\oplus$ operation constructs a value that is equivalent to the left argument. Intuitively, the map operation creates a monadic value with the same structure as the original. The law specifies that $\oplus$ prefers values from the left argument when both arguments have the same structure. The *left bias* law is different than the *left catch* law that is required about MonadOr in the MonadPlus reform proposal [11]. We return to this topic in Section 8.2, which discusses proposals for Haskell libraries.

6.3 MonadZip and MonadOr relation

Perhaps the most interesting law is required by the *distributivity* (9) equivalence. The law relates the $\oplus$ operation with $\otimes$ and fits very nicely with the rest of the theory:

$$a \otimes (b \oplus c) \equiv (a \otimes b) \oplus (a \otimes c) \qquad \text{(distributivity)}$$

This simple formulation does not hold when duplicating reference to a also duplicates effects and $\oplus$ is not able to undo the effects. For **docase**, the value a is always created by $malias$, so we could require a weaker law (that passes a to $malias$ first). We prefer the stronger version above for its simplicity. Thanks to the symmetry of $\otimes$, the law above also implies right distributivity.

6.4 MonadAlias type class

This section identifies the $malias$ laws. The type of the operation is $m\ a \to m\ (m\ a)$ and we treat it as a way to represent aliasing of monadic computations. As discussed in Section 7.2, operations of this type have other uses (with different set of laws).

The number of laws is relatively high, because $malias$ needs to interact with all other monad and joinad operations in a particular way. The following three laws consider $mzero$ and $return$ and are implied by *binding equivalence* (1), *matching units* (6) and *argument noninterference* (5), respectively:

$$malias\ a \gg\!= id \equiv a \qquad \text{(join identity)}$$
$$malias\ (return\ a) \equiv return\ (return\ a) \qquad \text{(unit identity)}$$
$$malias\ mzero \equiv return\ mzero \qquad \text{(zero identity)}$$

In the first law, applying $malias$ to a monadic value of type $m\ a$ yields a value of type $m\ (m\ a)$. The law specifies that immediate binding and returning has the same meaning as the original computation[4]. The next laws specify that aliasing of pure computation or failed computation does not have any effect.

The next four laws consider $malias$ given as an argument to $\gg\!=$ together with a function constructed in some special way.

$$(malias\ m \gg\!= f) \otimes n \equiv malias\ m \gg\!= (\otimes\ n) \circ f$$
$$malias\ (malias\ m \gg\!= f) \gg\!= g \equiv malias\ m \gg\!= (g \circ f)$$
$$map\ (map\ f)\ (malias\ m) \equiv malias\ (map\ f\ m)$$
$$map\ swap\ (malias\ m \star malias\ n) \equiv malias\ n \star malias\ m$$

$$\textbf{where}\ m \star n = m \gg\!= \lambda x \to n \gg\!= \lambda y \to return\ (x, y)$$

The first two laws are required by *distributivity* (9) to deal with nested aliasing and zipping of an aliased computation. The third law is implied by *matching images* (7) to lift the map operation over aliasing and the last law is required for *binding equivalence* (1) to reorder independent aliasing.

7. Theory of joinads

This section looks at categorical foundations of joinads and consider an algebraic structure formed by $\otimes$ and $\oplus$.

7.1 Monoidal functors

The discussion about MonadZip and Applicative from Section 6.1 can be recast in terms of category theory, because an Applicative instance corresponds to a monoidal functor. Given a monad, we can construct a monoidal functor. The MonadZip type class with the laws given above corresponds to a *symmetric monoidal functor* and $\otimes$ is the natural transformation defined by it. This is another justification for the *naturality*, *associativity* and *symmetry* laws.

Joinads combine this symmetric monoidal functor with a monad and thus also a monoidal functor specified by the monad. The underlying functor is the same, but the natural transformation and units differ. The unit of the $\otimes$ operation is not needed by joinads, so we do not require users to define it. In particular, this means that $return$ does not behave as unit with respect to $\otimes$. For example, a unit for List is a singleton list, but unit for ZipList is an infinite list. Zipping a list with an infinite list and then projecting out first elements of a tuple gives the original list, but the same is not true for zipping with a singleton list.

7.2 Computational comonads

The type of the $malias$ operation is the same as the type signature of $cojoin$ operation of a comonad. Although less frequent than monads, comonads are also a useful notion of computations in functional programming [23, 34], so it is worth considering how they relate to joinads. Comonads can be defined in multiple (equivalent) ways. The definition that uses $cojoin$ extends Functor with two operations and is shown in Figure 7. The $coreturn$ operation is dual to $return$ of a monad and $cojoin$ is dual to monadic $join$ which has the type $m\ (m\ a) \to m\ a$.

[4] The law can be reformulated using monadic join as $join(malias\ a) \equiv a$

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Comonad}\ m\ \textbf{where}\\ \quad \mathit{cojoin} :: m\ a \to m\ (m\ a)\\ \quad \mathit{coreturn} :: m\ a \to a\end{array}$$

Figure 7. Definition of Comonad type class.

The *cojoin* operation of a comonad could be used as the basis of *malias*, although joinads do not need the rest of the comonadic structure (the *coreturn* operation). We would only consider comonad laws that do not involve *coreturn*:

$$\mathit{map}\ (\mathit{map}\ f)\ (\mathit{cojoin}\ a) \equiv \mathit{cojoin}\ (\mathit{map}\ f\ a)$$
$$\mathit{map}\ \mathit{cojoin}\ (\mathit{cojoin}\ a) \equiv \mathit{cojoin}\ (\mathit{cojoin}\ a)$$

The first law is, indeed, one of the laws that we require to hold about the *malias* operation. The second law is not required to prove the equivalences from Figure 6, so we did not include it. However, it could be added and it should intuitively hold for *malias*.

Furthermore, *computational comonads* introduced by Brookes and Geva [3] are even more closely related to joinads. A computational comonad is a comonad (T, ϵ, δ) with an additional *natural transformation* $\gamma : I_C \to T$. In terms of Haskell, this is a function of type $a \to m\ a$ and it coincides with the *return* operation of a joinad. Computational comonad has to satisfy the usual laws of a comonad together with three additional operations that relate γ with *cojoin* and *coreturn*. We write *return* for γ:

$$\mathit{map}\ f\ (\mathit{return}\ a) \equiv \mathit{return}\ (f\ a)$$
$$\mathit{cojoin}\ (\mathit{return}\ a) \equiv \mathit{return}\ (\mathit{return}\ a)$$
$$\mathit{coreturn}\ (\mathit{return}\ a) \equiv a$$

The first law is a naturality law of the *return* operation that can be proved from the standard monad laws and therefore it holds for joinads. The second law corresponds to the *unit identity* law that we also require about joinads (it is required by the *matching units* (6) transformation). Finally, the third law of computational comonads involves the *coreturn* operation that is not present in joinads, so it is not directly relevant. We find this close correspondence intriguing and intend to explore it in a future work.

7.3 Joinad algebra

The laws about $\otimes$ and $\oplus$ discussed in the previous section suggest that joinads can be modelled as an algebraic structure. Assume that J is a set containing all monadic values of some type $m\ a$ for the same joiand m. The *mzero* value is a special element $0 \in J$.

- The $\oplus$ operation is associative and has 0 as the identity element. This means that the structure $(J, \oplus, 0)$ is a monoid.
- The $\otimes$ operation is commutative and associative, which means that $(J, \otimes)$ is a commutative semigroup. Additionally, the semigroup has 0 as the zero element.
- Finally, the $\otimes$ operation distributes over the $\oplus$ operation.

These axioms characterize an algebraic structure called *near-semiring* with commutative $\otimes$ operation. This specification captures the essence of joinads—the only thing that is left out is the left bias of the $\oplus$ operation. As discussed in Section 9.1 this general case may be also useful, but we require left bias so that **docase** has semantics inspired by the Haskell's **case** construct.

8. Feature interactions and library proposals

Joinads combine type classes that are already known to the Haskell community. This section considers adjustments that could be made to MonadZip and MonadOr in order to accommodate joinads.

8.1 Monad comprehension laws

There is an ongoing discussion about the laws that should be required for parallel monad comprehensions [1, 9]. The original documentation specified the following two laws about MonadZip:

$$\mathit{map}\ f\ a \otimes \mathit{map}\ g\ b \equiv \mathit{map}\ (f \times g)\ (a \otimes b) \qquad \text{(naturality)}$$
$$\mathit{map}\ \mathit{fst}\ (a \otimes b) \equiv a \qquad \text{(information preservation)}$$

Monoidal laws. The *naturality* law was also proposed for joinads (Section 6.1). It arises from the syntactic equivalences, but also from the fact that the *mzip* operation is defined by a monoidal functor. We propose the following two additions:

- The *associativity* law also arises from a monoidal functor, hence the two should be both required.
- Symmetry is an essential aspect of *mzip* and we argue that the *symmetry* law should be also included in MonadZip.

The symmetry of *mzip* holds for lists as well as for the MonadZip instances presented in this paper. In terms of parallel monad comprehensions, the law guarantees the following equivalence:

$$[(a, b) \mid a \leftarrow m_1 \mid b \leftarrow m_2] \equiv [(a, b) \mid b \leftarrow m_2 \mid a \leftarrow m_1]$$

The symmetry means that the *mzip* operation cannot be automatically implemented in terms of $\gg\!=$ and *return*. This specifies that the additional syntax should also have an additional meaning. It is still possible to get *mzip* for free, but only for *commutative monads*, which is discussed in earlier work on joinads [27].

Information preservation. The second law specifies that we can recover the original arguments of a value created using $\otimes$. This law is problematic. It allows applying *mzip* to inputs with different structure (i.e. length of the list), but recovering the original values is only possible if the structure of arguments is the same. For example, *zip* for lists restricts the length to the length of the shorter lists, so the original law does not hold for lists of different length.

Using *naturality* and the *duplication* law, we can derive the following law that looks similar and clarifies the requirement about the structure of values:

$$\mathit{map}\ \mathit{fst}\ (a \otimes\ \mathit{map} f a) \equiv a \equiv \mathit{map}\ \mathit{snd}\ (\mathit{map}\ g\ a\ \otimes\ a)$$

Instead of zipping two arbitrary monadic values, the law zips a value with an image created using *map*. Thanks to the properties of *map*, the law only concerns zipping of monadic values with the same structure. Hence, we make the following proposal:

- The *information preservation* law does not hold for many standard implementations of *mzip*, so we propose replacing it with the weaker form presented above.

The *product* and *zero* laws can be also translated in terms of parallel monad comprehensions, but we do not find them as essential.

8.2 Left-biased additive monads

Monads that are also monoids and provide an *mzero* element and an associative $\oplus$ operation are captured by the MonadPlus type class from the standard Haskell library. However, there is some confusion about the additional laws that should hold. The MonadPlus reform proposal [11] provides a solution by splitting the type class into MonadPlus obeying the *left distribution* law and MonadOr obeying the *left catch* law. The *left bias* law that we require for joinads (Section 6.2) adds a third alternative:

$$u \oplus v \gg\!= f \equiv (u \gg\!= f) \oplus (v \gg\!= f) \qquad \text{(left distribution)}$$
$$(\mathit{return}\ a) \oplus u \equiv \mathit{return}\ a \qquad \text{(left catch)}$$
$$u \oplus \mathit{map}\ f\ u \equiv u \qquad \text{(left bias)}$$

$$
\begin{array}{l}
\textbf{instance } \mathit{MonadOr}\ [\,]\ \textbf{where} \\
\quad \mathit{morelse}\ (x:xs)\ (y:ys) = x : \mathit{morelse}\ xs\ ys \\
\quad \mathit{morelse}\ [\,]\ ys = ys \\
\quad \mathit{morelse}\ xs\ [\,] = xs \\
\textbf{instance } \mathit{MonadOr}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{morelse}\ (\mathit{Just}\ a)\ \_ = \mathit{Just}\ a \\
\quad \mathit{morelse}\ \mathit{Nothing}\ b = b
\end{array}
$$

Figure 8. Instance of MonadOr that obey left bias law

It is not difficult to find counter-examples showing that none of the three laws implies some other. Both *left bias* and *left catch* represent some form of left bias, but in a different way.

- The *left bias* law uses an arbitrary value as the left and a special value (constructed using map) as the right argument.
- The *left catch* law uses an arbitrary value as the right and a special value (constructed using $unit$) as the left argument.

Despite the difference, the main purpose of the two laws is the same. They both specify that the operation is left biased. Which law should hold about MonadOr? One option is to consider the upper or the lower bound of the two laws:

$$(return\ a) \oplus (return\ b) \equiv return\ a \qquad \text{(lower bound)}$$
$$u \oplus u \gg\!= f \equiv u \qquad \text{(upper bound)}$$

The *upper bound* implies by both left bias and left catch, while the *lower bound* is implied by any of the two. It is not clear to us whether any monad can provide a non-trivial implementation of $\oplus$ satisfying the *upper bound* law. The *lower bound* law is more appropriate, although it is not sufficient to prove that the *clause ordering* equation from Section 5 holds.

We argue that *left bias* better captures the purpose. The most prominent monad that satisfies MonadOr laws is the Maybe monad, which obeys both of the laws. We can also give a useful implementation of $morelse$ that obeys the left bias law for the List monad. The two declarations are shown in Figure 8. An alternative would be to separate the type from the laws that are required in the language, but this is a separate research topic.

9. Related and future work

We presented an earlier version of joinads in F$^{\#}$ [27] using different examples. An article for The Monad.Reader [25] provides more details on the relation between joinads and *monad comprehensions*.

The rest of this section presents some of the important related work on pattern matching, concurrent programming and abstract computation types as well as preliminary ideas for future work.

9.1 Backtracking and committing patterns

Existing work on pattern matching has focused on enabling pattern matching on abstract values using views [35]. A similar concept also appeared in F$^{\#}$ and Scala [7, 31]. Making patterns first-class made it possible to encode the Join calculus using Scala [10], although the encoding is somewhat opaque.

Some authors [31, 33] have suggested generalizing the result of pattern matching from Maybe (representing a failure or a success) to any additive monad using the MonadPlus type class. The concrete examples included encoding of backtracking using the List monad and composing transactions using STM.

The next example demonstrates the difference between joinads as described here and the MonadPlus interpretation. Assuming standard parser combinators ($char$, $many$, and $item$), we can write:

$$
\begin{array}{l}
body = \textbf{mcase}\ (char\ \texttt{'('}, many\ item)\ \textbf{of} \\
\quad (\_, ?) \quad \to \textbf{do}\ str \leftarrow body \\
\qquad\qquad\qquad\quad \_ \leftarrow char\ \texttt{')'} \\
\qquad\qquad\qquad\quad return\ str \\
\quad (?, str) \to return\ str
\end{array}
$$

The **mcase** construct (similar to our **docase**) represents a monadic pattern matching using MonadPlus. In the syntax designed for active patterns [31], monadic values were produced by *active patterns* that return monadic values. For parsers, the type of active patterns would be $a \to Parser\ b$. This leads to a different syntax, but it is possible to translate between the two options.

The translation based on MonadPlus follows similar pattern as the translation of joinads, but differs in three ways:

$$
\begin{array}{l}
(char\ \texttt{'('} \gg\!= \backslash\_ \to body \gg\!= \lambda str \to \\
\quad char\ \texttt{')'} \gg\!= \backslash\_ \to return\ str) \ \oplus \\
(many\ item \gg\!= \lambda str \to return\ str)
\end{array}
$$

The first difference (apparent from this example) is that the proposed encoding using MonadPlus does not add additional wrapping around the body of the alternatives to support committing to an alternative. The second difference is that MonadPlus usually requires the *left distributivity* law instead of the *left bias* law required by MonadOr. Finally, multiple binding patterns are translated using nested $\gg\!=$ instead of a special $mzip$ operation.

- When using MonadOr, the $\oplus$ operation attempts to parse the input using the first alternative. Other alternatives are considered only if the first one fails. The above parser would deterministically parse "((1))" as "1". The laws of MonadPlus make the resulting parser non-deterministic, so it would generate three options: "1", "(1)" or "((1))".
- Without the additional wrapping, the parser needs to implement backtracking. If the input is "(1", the first alternative is selected, continues consuming input, but then fails. The parser needs to backtrack to the point when $\oplus$ was used and try the second alternative. When using wrapping, the parser will commit to the first alternative, which is an approach used by modern libraries such as Parsec [19].
- Combining multiple inputs using the $mzip$ operation means that the arguments of **docase** can be reordered even for non-commutative monads. A separate $mzip$ operation may also enable additional optimizations, for example, in the STM monad.

The example above shows that all of the options may have a feasible meaning for some monads. We find the joinad-based semantics of **docase** that supports commit points more appropriate for monads from functional programming. The variant using MonadPlus often implies backtracking and thus may be more suitable for logic programming languages such as Prolog.

9.2 Commit points in remote procedure calls

The discussion about commit points in Section 3.4 was inspired by Concurrent ML (CML) [28]. CML is a concurrent programming language built on top of Standard ML. It supports first-class synchronization values called *events* that can be used to encode many common concurrent programming patterns. Joinads, on the other hand, capture a single pattern that we find extremely important.

We demonstrate the relation between joinads and CML, by showing two implementations of a remote procedure call (RPC). Assume we have a monad for blocking communication and monadic computations $send$, $recv$ that represent sending request and receiv-

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Monoidal}\ f \textbf{ where} \\
\quad \mathit{unit} :: f\ () \\
\quad (\star) :: f\ a \to f\ b \to f\ (a, b) \\
\textbf{class } \mathit{Monoidal}\ f \Rightarrow \mathit{Alternative}\ f \textbf{ where} \\
\quad \mathit{empty} :: f\ a \\
\quad (\diamond) :: f\ a \to f\ a \to f\ a
\end{array}
$$

Figure 9. Alternative type class

ing a response. As an alternative to performing the RPC call, the client can choose to perform another operation alt.

One way to implement the RPC call is to initiate a call, but allow abandoning the RPC communication at any time until it completes. This means that receiving a response from the server is used as a commit point for the RPC call:

$$
\begin{array}{l}
\textbf{docase}\ (\mathit{send}, \mathit{recv}, \mathit{alt})\ \textbf{of} \\
\quad ((), \mathit{res}, ?) \to \mathit{handleRpc}\ \mathit{res} \\
\quad (?, ?, a) \quad\ \to \mathit{handleAlt}\ a
\end{array}
$$

We can assume that the event $recv$ becomes enabled after $send$, so the first alternative becomes enabled after the server replies. The second alternative will be selected if the alt event is enabled earlier. This may, or may not, happen before the server accepts the request and enables the $send$ event.

The second way to implement RPC is to allow abandoning the communication only before the server accepts the request. After that, the client waits for $recv$ event and cannot choose alt instead:

$$
\begin{array}{l}
\textbf{docase}\ (\mathit{send}, \mathit{alt})\ \textbf{of} \\
\quad ((), ?) \to \textbf{do}\ \mathit{res} \leftarrow \mathit{recv} \\
\qquad\qquad\quad\ \ \mathit{handleRpc}\ \mathit{res} \\
\quad (?, a)\ \ \to \mathit{handleAlt}\ a
\end{array}
$$

In this version of code, the **docase** construct only chooses between $send$ and alt. Once the first alternative is selected, it has to wait for the server response using $recv$.

This section demonstrates that joinads can capture the two essential patterns for writing RPC communication as introduced in Concurrent ML. This example critically relies on the support for commit points introduced in Section 3.4. When using the simple encoding discussed in Section 9.1, the two expressions would translate to the same meaning.

9.3 Joinads and other computation types

Joinads extend monads to support the **docase** construct, but functional languages use several other notions of computation. In the future, it may be interesting to consider how other computations relate to generalized pattern matching. Comonads (a categorical dual of monads) [17] have been used for encoding data-flow programs [34], but also for stencil computations using special *grid patterns* [23]. Arrows [13, 20] are used mainly in functional reactive programming research [12] and can be written using the arrow notation [24] (in a similar way to how monads use the **do** notation).

Another notion of computation is called *applicative functors* [22] or *idioms*, which are weaker than monads and can thus capture larger number of computations. Haskell libraries also include an Alternative type class that extends applicative functors with a $\diamond$ operator similar to $\oplus$ from MonadPlus or MonadOr. The declaration is shown in Figure 9. The figure shows Monoidal type class, which is equivalent to a more common Applicative class (as discussed by McBride and Patterson [22] in Section 7), because this variation better reveals similarity with joinad operations.

Interestingly, the operations of Alternative have the same types as the two most essential operations of joinads. The $\star$ operation has the same type as our $mzip$, representing parallel composition, and $\diamond$ has the type of $morelse$, representing a choice. It is well known that applicative functors are more general than monads and Alternative may generalize joinads in a similar way.

9.4 Applications

We demonstrated that **docase** notation can be used for working with many common monads. When using monadic parser combinators [14] the $morelse$ operation represents left-biased choice as supported in [19]. As discussed in our earlier article, our implementation of parallel composition (the $mzip$ operation) corresponds to the intersection of context-free grammars [25]. We are not aware of any parser combinator library that provides this operation, but it seems to be very useful for validation of inputs (eliminating inputs that do not match any of the given parsers).

The reactive programming examples used in this paper were based on imperative streams developed by Scholz [29]. Imperative streams are essentially monads for synchronous reactive programming. A Push-Pull Functional Reactive Programming framework developed by Elliott [6] includes a monad instance for events, so it could likely benefit from the **docase** syntax too.

The parallel programming model that we presented can be added to various existing Haskell frameworks. Our earlier article [25] used strategies [21]. In this paper, we embedded the examples in the Par monad [15] with several extensions to allow speculative computations [26]. The programming model is very similar to the **pcase** construct provided by Manticore [8].

10. Conclusions

This paper presented a characterization of monadic computations that provide three additional operations: *aliasing*, *parallel composition* and *choice*. These operations are not new to Haskell. They are captured by type classes Extend, MonadZip and MonadOr. We combined them and designed a **docase** notation that makes it easy to compose computations using these operations.

The **docase** notation is inspired by our previous work on joinads in F$^{\#}$. However, this paper uses a simpler set of operations that are amenable to formal reasoning. We started with a set of semantics-preserving transformations that are intuitively expected to hold about the **docase** construct. We derived a set of laws about joinad operations and used the Coq theorem prover to show that these suffice to perform the intuitive transformations. We also noted that joinads form an algebraic structure known as *near-semiring*.

Finally, we also made several concrete library proposals based on our work. In particular, we support the proposal to distinguish between unbiased MonadPlus and left-biased MonadOr and we propose a refined set of laws that should be required by MonadZip. We demonstrated the usefulness of our extension using a wide range of monads including reactive and parallel programming as well as input validation using monadic parsers.

Acknowledgments

We are grateful to Philip Wadler for encouraging feedback about earlier versions of this work and to Dominic Orchard for many useful discussions about this paper and Haskell in general. Simon Peyton Jones, Gregory Neverov, Dmitry Lomov and James Margetson all provided useful comments about early work on joinads. We also thank to anonymous referees for useful comments.

References

[1] Haskell Trac. Bring back monad comprehensions, retrieved 2011. `http://hackage.haskell.org/trac/ghc/ticket/4370`.

[2] Haskell Documentation. Control.Monad, retrieved 2011

[3] S. Brookes, S. Geva. Computational comonads and intensional semantics. Technical Report CMU-CS-91-190, Carnegie Mellon University, 1991.

[4] N. C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency using Monads. In *Communicating Process Architectures 2008*, pages 67–83, 2008.

[5] K. Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9:313–323, May 1999. ISSN 0956-7968.

[6] C. Elliott. Push-Pull Functional Reactive Programming. In *Haskell Symposium*, 2009.

[7] B. Emir, M. Odersky, and J. Williams. Matching Objects with Patterns. In *ECOOP 2007*, 2007.

[8] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20 (Special Issue 5-6):537–576, 2010.

[9] G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers Bringing Back Monad Comprehensions. To appear in *Haskell'11*, 2011

[10] P. Haller and T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, 2008.

[11] HaskellWiki. MonadPlus reform proposal, retrieved 2011. `http://tinyurl.com/monadplus-reform-proposal`.

[12] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, volume 2638 of *LNCS*, pages 1949–1949. 2003.

[13] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, 1998.

[14] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[15] S. P. Jones, S. Marlow, and R. Newton. A Monad for Deterministic Parallelism, 2011. `http://tinyurl.com/monad-par`.

[16] S. P. Jones and P. Wadler. Comprehensive Comprehensions. In *Haskell'07*, pages 61–72, 2007.

[17] R. B. Kieburtz. Codata and Comonads in Haskell. Unpublished manuscript, 1999.

[18] E. A. Kmett. The comonad package, retrieved 2011. `http://hackage.haskell.org/package/comonad`.

[19] D. Leijen and E. Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[20] H. Liu, E. Cheng, and P. Hudak. Causal Commutative Arrows and Their Optimization. In *ICFP'09*, pages 35–46, 2009.

[21] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: Better Strategies for Parallel Haskell. In *Haskell'10*.

[22] C. McBride and R. Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18:1–13, 2007.

[23] D. A. Orchard, M. Bolingbroke, and A. Mycroft. Ypnos: Declarative, Parallel Structured Grid Programming. DAMP '10, 2010.

[24] R. Paterson. A New Notation for Arrows. In Proceedings of *ICFP'01*, pages 229–240. ACM Press, Sept. 2001.

[25] T. Petricek. Fun with parallel monad comprehensions, 2011. The Monad.Reader, Issue 18.

[26] T. Petricek. Explicit speculative parallelism for Haskell's Par monad, `http://tomasp.net/blog/speculative-par-monad.aspx`, retrieved 2011.

[27] T. Petricek and D. Syme. Joinads: A retargetable control-flow construct for reactive, parallel and concurrent programming. In *PADL'11*, pages 205–219, 2011.

[28] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007. ISBN 978-0-521-71472-3.

[29] E. Scholz. Imperative Streams—A Monadic Combinator Library for Synchronous Programming. In ICFP'98, 1998.

[30] S. D. Swierstra. Combinator Parsing: A Short Tutorial. Technical report, Utrecht University, 2008.

[31] D. Syme, G. Neverov, and J. Margetson. Extensible Pattern Matching via a Lightweight Language Extension. ICFP, 2007.

[32] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.

[33] M. Tullsen. First-Class Patterns. In *PADL 2000*, 2000.

[34] T. Uustalu and V. Vene. The essence of dataflow programming. In *APLAS*, pages 2–18, 2005.

[35] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. POPL, 1987.

[36] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990

A. Explicit shortcircuiting

The motivating example in Section 2 used **docase** and the Par monad to implement a *all* function for trees. The function takes a predicate and tests whether it holds for all values stored the tree.

The following code implements the functionality *Par* monad with an extension that allows cancellation of tasks [26]. This code does not represent desugared version of the **docase** notation. Instead, it represents a typical solution that developers may write when using the library directly:

```
all :: (a → Bool) → Tree a → Par Bool
all p tree = do
    tok ← newCancelToken
    r ← all′ tok tree
    cancel tok
    return r
  where
    all′ tok (Leaf v) = return (p v)
    all′ tok (Node left right) = do
      leftRes ← new
      rightRes ← new
      finalRes ← newBlocking
      forkWith tok (all′ tok left ≫=
        completed leftRes rightRes finalRes)
      forkWith tok (all′ tok right ≫=
        completed rightRes leftRes finalRes)
      get finalRes
    completed varA varB fin resA = do
      put varA resA
      if ¬ resA then put fin False
      else get varB ≫= put fin ∘ (∧ resA)
```

The main function creates a new cancellation token and then calls a helper that does the processing. The cancellation token is used to stop all pending computations when the overall result is known.

Inside *all′*, the variables *leftRes* and *rightRes* are used to store the result of their corresponding computations. The last variable is created differently: when the variable is full and a computation attempts to write into it, it will block instead of failing. The *all′* function then spawns two tasks to process sub-trees and waits for the final result.

The two computations both make a recursive call and then pass the result to *completed*. If the result is *False*, the function sets the final result. Otherwise, it waits until the other computation completes and then calculates the final result.