Evaluation strategies for monadic computations

Tomas Petricek
Computer Laboratory
University of Cambridge
United Kingdom
tomas.petricek@cl.cam.ac.uk

Monads have become a powerful tool for structuring effectful computations in functional programming, because they make the order of effects explicit. When translating pure code to a monadic version, we need to specify evaluation order explicitly. This requires us to choose between *call-by-value* or *call-by-name* style. The two translations give programs with different semantics, structure and also different types.

In this paper, we translate pure code to monadic using an additional operation *malias* that abstracts out the evaluation strategy. The *malias* operation is based on *computational comonads*; we use a categorical framework to specify the laws that are required to hold about the operation.

We show two implementations of *malias* for any monad that give *call-by-value* and *call-by-name* semantics. Although we do not give *call-by-need* semantics for *any* monad, we show how to turn any monad into an extended monad with *call-by-need* semantics, which partly answers a standing open question. Moreover, using our unified translation, it is possible to change the evaluation strategy of functional code translated to the monadic form without changing its structure or types.

1 Introduction

Purely functional languages use lazy evaluation (also called *call-by-need*) to allow elegant programming with infinite data structures and to guarantee that a program will not evaluate a diverging term unless it is needed to obtain the final result. However, reasoning about lazy evaluation is difficult and so lazy evaluation is not suitable for programming with effects.

An elegant way to embed effectful computations in lazy functional languages is to use monads introduced by Moggi [16] and Wadler [24]. Using monads, we can translate purely functional code to a monadic version that explicitly specifies the evaluation order. Wadler [24] gives two ways of translating pure programs.

One approach leads to a *call-by-value* semantics, where effects of function arguments are performed before calling a function. However, if an argument has an effect and kills the program, this may not be appropriate if the function can successfully complete without using the argument. Second approach gives a *call-by-name* semantics, where effects are performed only if argument is actually used. However, this approach is not always suitable either, because an effect may be performed repeatedly. Wadler leaves an open question whether there is a translation that would correspond to *call-by-need* semantics, where effects are performed only when the result is needed, but at most once.

The main contribution of this paper is an alternative translation of functional code to a monadic form, parameterized by an operation *malias*. It answers the above open question for certain monads and has the following desirable properties:

- A single translation gives monadic code with either call-by-name or call-by-value semantics, depending on the definition of *malias* (Section 2). When used in language such as Haskell, it is possible to write code that is parameterized by the evaluation strategy (Section 4.1).

Submitted to:
MSFP 2012

- We answer the open question posed by Wadler and show how to use our translation to write monadic computations with *call-by-need* semantics (Section 4.2). Furthermore, for some monads, it is possible to use *parallel call-by-need* semantics, where arguments are evaluated in parallel with the body of a function (Section 4.3).
- The *malias* operation has solid foundations in category theory. It arises from combining a *monad* with a *computational semi-comonad* based on the same functor (Section 3). We use this theory to define laws that should be obeyed by *malias* implementations (Section 2.2).

This paper was inspired by work on *joinads* [19], which introduced the *malias* operation for a similar purpose. However, operations with the same type and similar laws appear several times in the literature. We return to joinads in Section 4.4 and review other related work in Section 5.

1.1 Translating to monadic code

We first demonstrate the two standard options for translating purely functional code to monadic form. Consider the following two functions that use *lookupInput* to read some configuration property. Assuming the configuration is already loaded in memory, we can write the following pure computation:

$$
\begin{array}{l}
\mathit{chooseSize} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
\mathit{chooseSize}\ \mathit{new}\ \mathit{legacy} = \\
\quad \mathbf{if}\ \mathit{new} > 0\ \mathbf{then}\ \mathit{new}\ \mathbf{else}\ \mathit{legacy} \\
\mathit{resultSize} :: \mathit{Int} \\
\mathit{resultSize} = \\
\quad \mathbf{let}\ \mathit{new} = \mathit{lookupInput}\ \texttt{"new\_size"} \\
\quad\quad\ \ \mathit{legacy} = \mathit{lookupInput}\ \texttt{"legacy\_size"} \\
\quad \mathbf{in}\ \ \mathit{chooseSize}\ \mathit{new}\ \mathit{legacy}
\end{array}
$$

The *resultSize* function reads two different configuration keys and then chooses one of them using *chooseSize*. Thanks to the lazy evaluation, the call *lookupInput* `"legacy_size"` is performed only when the value of `"new_size"` is less than or equal to zero.

To modify the function to actually read configuration from a file as opposed to performing in-memory lookup, we first change *lookupInput* to return *IO Int* instead of *Int*. Then we need to modify the two above functions. There are two mechanical ways that give different semantics.

Call-by-value. In the first style, we call *lookupInput* and then apply monadic bind on the resulting computation. This reads both of the configuration values before calling the *chooseSize* function, and so arguments are fully evaluated before the body of a function as in the *call-by-value* evaluation strategy:

$$
\begin{array}{l}
\mathit{chooseSize}_{cbv} :: \mathit{Int} \to \mathit{Int} \to \mathit{IO}\ \mathit{Int} \\
\mathit{chooseSize}_{cbv}\ \mathit{new}\ \mathit{legacy} = \\
\quad \mathit{return}\ (\mathbf{if}\ \mathit{new} > 0\ \mathbf{then}\ \mathit{new}\ \mathbf{else}\ \mathit{legacy}) \\
\mathit{resultSize}_{cbv} :: \mathit{IO}\ \mathit{Int} \\
\mathit{resultSize}_{cbv} = \mathbf{do} \\
\quad \mathit{new} \leftarrow \mathit{lookupInput}_{cbv}\ \texttt{"new\_size"} \\
\quad \mathit{legacy} \leftarrow \mathit{lookupInput}_{cbv}\ \texttt{"legacy\_size"} \\
\quad \mathit{chooseSize}_{cbv}\ \mathit{new}\ \mathit{legacy}
\end{array}
$$

In this version of the translation, a function of type $A \to B$ is turned into a function $A \to M\ B$. For example, the $chooseSize_{cbv}$ function takes integers as parameters and returns a computation that returns an integer and may perform some effects. When calling a function in this setting, the arguments may not be fully evaluated (the functional part is still lazy), but the effects associated with obtaining the value of the argument happen before the function call.

For example, if the call $lookupInput_{cbv}$ `"new_size"` read a file and then returned 1024, but the operation $lookupInput_{cbv}$ `"legacy_size"` caused crash of the program because a specified key was not present in a configuration file, then the entire program would crash.

Call-by-name. In the second style, we pass unevaluated computations as arguments to functions. This means we call *lookupInput* to create an effectful computation that will read the input, but the computation is then passed to *chooseSize*, which may not need to evaluate it:

$$
\begin{array}{l}
chooseSize_{cbn} :: IO\ Int \to IO\ Int \to IO\ Int \\
chooseSize_{cbn}\ new\ legacy = \mathbf{do} \\
\quad newVal \leftarrow new \\
\quad \mathbf{if}\ newVal > 0\ \mathbf{then}\ new\ \mathbf{else}\ legacy
\end{array}
$$

$$
\begin{array}{l}
resultSize_{cbn} :: IO\ Int \\
resultSize_{cbn} = \\
\quad \mathbf{let}\ new = lookupInput_{cbn}\ \texttt{"new\_size"} \\
\quad\quad legacy = lookupInput_{cbn}\ \texttt{"legacy\_size"} \\
\quad \mathbf{in}\ chooseSize_{cbn}\ new\ legacy
\end{array}
$$

The translation turns a function of type $A \to B$ into a function $M\ A \to M\ B$. This means that the $chooseSize_{cbn}$ function takes a computation that performs the I/O effect and reads information from the configuration file, as opposed to taking a value whose effects were already performed.

Following the mechanical translation, $chooseSize_{cbn}$ returns a monadic computation that evaluates the first argument and then behaves either as *new* or as *legacy*, depending on the obtained value. When the resulting computation is executed, the computation *new* (that reads value of the `"new_size"` key) may be executed repeatedly. In this particular example, we can easily change the code to perform the effect just once, but there is no general way of doing it, so the computation gives the *call-by-name* semantics.

2 Abstracting evaluation strategy

The translations demonstrated in the previous section have two major problems. Firstly, it is not easy to switch between the two – when we introduce effects using monads, we need to decide to use one or the other style and changing between them later on involves rewriting of the program and changing types. Secondly, even in the *IO* monad, we cannot easily implement a *call-by-need* strategy that would perform effects only when a value is needed, but at most once.

2.1 Translation using aliasing

To solve these problems, we propose an alternative translation. We require a monad m with an additional operation *malias* that abstracts out the evaluation strategy and has a type $m\ a \to m\ (m\ a)$. Before looking at various implementation of *malias*, we give a new translation of the previous example:

$$
\begin{array}{l}
\mathit{chooseSize} :: \mathit{IO\ Int} \to \mathit{IO\ Int} \to \mathit{IO\ Int} \\
\mathit{chooseSize\ new\ legacy} = \mathbf{do} \\
\quad \mathit{newVal} \leftarrow \mathit{new} \\
\quad \mathbf{if}\ \mathit{newVal} > 0\ \mathbf{then}\ \mathit{new}\ \mathbf{else}\ \mathit{legacy} \\
\mathit{fileSize} :: \mathit{IO\ Int} \\
\mathit{fileSize} = \mathbf{do} \\
\quad \mathit{new} \leftarrow \mathit{malias}\ (\mathit{lookupInput}\ \texttt{"new\_size"}) \\
\quad \mathit{legacy} \leftarrow \mathit{malias}\ (\mathit{lookupInput}\ \texttt{"legacy\_size"}) \\
\quad \mathit{chooseSize\ new\ legacy}
\end{array}
$$

The types of functions and access to function parameters are translated in the same way as in the *call-by-name* translation. The *chooseSize* function returns a computation *IO Int* and its parameters also become computations of type *IO Int*. When using the value of a parameter, the computation is evaluated using monadic bind (e.g. the line $\mathit{newVal} \leftarrow \mathit{new}$ in *chooseSize*).

However, the computations passed as arguments to functions are not the original computations as in the *call-by-name* translation. Instead, the translation inserts a call to *malias* for every argument of function application or every let-bound value. The computation returned by *malias* has a type $m\ (m\ a)$, which makes it possible to perform the effects at two different call sites:

- When simulating the *call-by-value* strategy, all effects are performed when binding the outer monadic computation before a function call.
- When simulating the *call-by-name* strategy, all effects are performed when binding the inner monadic computation, when the value is actually needed.

These two strategies can be implemented for any monad and are surprisingly simple. However, by delegating the implementation of *malias* to the monad, we make it possible to implement more advanced strategies as well. We discuss some of them later in Section 4. We keep the translation informal until Section 2.3 and discuss the *malias* operation in more detail first.

Implementing call-by-name. To implement the *call-by-name* strategy, the *malias* operation needs to return the computation specified as an argument inside the monad. In the type $m\ (m\ a)$, the outer m will not carry any effects and the inner m will be the same as the original computation:

$$
\begin{array}{l}
\mathit{malias} :: m\ a \to m\ (m\ a) \\
\mathit{malias}\ m = \mathit{return}\ m
\end{array}
$$

From the monad laws (cf. Figure 1), we know that binding on a computation by applying *return* to some value is equivalent to just passing the value to the rest of the computation. This means that the additional binding in the translation does not have any effect and the resulting program behaves as the *call-by-name* strategy. A complete proof can be found in Appendix A.

Implementing call-by-value. Implementing the *call-by-value* strategy is similarly simple. The returned computation needs to perform all the effects when binding the outer m in the type $m\ (m\ a)$ the inner m will be just a computation that returns the computed value without performing any effects:

$$
\begin{array}{l}
\mathit{malias} :: m\ a \to m\ (m\ a) \\
\mathit{malias}\ m = m \gg\!= (\mathit{return} \circ \mathit{return})
\end{array}
$$

Functor with unit and join:

$$\begin{aligned}
&unit :: a \to m\ a\\
&map :: m\ a \to (a \to b) \to m\ b\\
&join :: m\ (m\ a) \to m\ a
\end{aligned}$$

$$\begin{aligned}
&join \circ map\ join = join \circ join\\
&join \circ map\ return = id = join \circ return\\
&join \circ map\ (map\ f) = map\ f \circ join
\end{aligned}$$

Using bind and return:

$$\begin{aligned}
&return :: a \to m\ a\\
&\gg= \ :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}$$

$$\begin{aligned}
&return\ a \gg= f \equiv f\ a\\
&m \gg= return \equiv m\\
&(m \gg= f) \gg= g \equiv m \gg= (\lambda x \to f\ x \gg= g)
\end{aligned}$$

Figure 1: Two equivalent ways of defining monads with monad laws

In Haskell, the second line could be written as *liftM return m*. The *liftM* operation represents the functor associated with the monad. This means that binding on the returned computation performs all the effects, obtains a value v and returns a computation *return v*.

When calling a function that takes an argument of type $m\ a$, the argument passed to it using this implementation of *malias* will always be constructed using the *return* operation. Hence the resulting behaviour is equivalent to the original *call-by-value* translation. Detailed proof can be found in Appendix A.

2.2 The malias operation laws

In order to define a reasonable evaluation strategy, we require the *malias* operation to obey a number of laws. The laws follow from the theoretical background that is discussed in Section 3, namely from the fact that *malias* is the *cojoin* operation of a computational semi-comonad.

The laws that relate *malias* to the monad are easier to write in terms of *join*, *map* and *unit* than using the formulation based on $\gg=$ and *return*. For completeness, the two equivalent definitions of monads with the monad laws are shown in Figure 1. Although we do not show it, one can be easily defined in terms of the other. The required laws for *malias* are the following:

$$\begin{aligned}
map\ (map\ f) \circ malias &= malias \circ (map\ f) && (naturality)\\
map\ malias \circ malias &= malias \circ malias && (associativity)\\
malias \circ unit &= unit \circ unit && (computationality)\\
join \circ malias &= id && (identity)
\end{aligned}$$

The first two laws follow from the fact that *malias* is a *cojoin* operation of a comonad. The *naturality* law specifies that applying function to a value inside a computation is the same as applying value to an aliased computation inside a computation. The *associativity* law specifies that aliasing an aliased computation is the same as aliasing a computation produced by an aliased computation.

The *computationality* law is derived from the fact that the comonad defining *malias* is *computational comonad* with *unit* as one of the components. The law specifies that aliasing of a pure computation creates a pure computation.

The *identity* law specifies that *join* is a *left inverse* of *malias*. It relates the comonadic structure (the *malias* operation) with the monadic structure. Intuitively, it specifies that aliasing a computation of type $m\ a$ and then joining the result returns the original computation. The motivation for the law is the following – given a computation of type $m\ a$, the *malias* operation constructs a computation of type $m\ (m\ a)$. This is done by splitting the information (or effects) of the computation between two monadic computations. Requiring that applying *join* to the new computation returns the original computation

$$
\begin{array}{lcl}
[\![x]\!]_{cbn} & = & x \\
[\![\lambda x.e]\!]_{cbn} & = & unit\ (\lambda x.[\![e]\!]_{cbn}) \\
[\![e_1\, e_2]\!]_{cbn} & = & bind\ [\![e_1]\!]_{cbn}\ (\lambda f.f\ [\![e_2]\!]_{cbn}) \\
[\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]_{cbn} & = & (\lambda x.[\![e_2]\!]_{cbn})\ [\![e_1]\!]_{cbn}
\end{array}
$$

Figure 2: Standard call-by-name translation of λ calculus.

$$
\begin{array}{lcl}
[\![x]\!]_{cbv} & = & unit\ x \\
[\![\lambda x.e]\!]_{cbv} & = & unit\ (\lambda x.[\![e]\!]_{cbv}) \\
[\![e_1\, e_2]\!]_{cbv} & = & bind\ [\![e_1]\!]_{cbv}\ (\lambda f.\ bind\ [\![e_2]\!]_{cbv}\ (\lambda x.f\ x)) \\
[\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]_{cbv} & = & bind\ [\![e_1]\!]_{cbv}\ (\lambda x.[\![e_2]\!]_{cbv})
\end{array}
$$

Figure 3: Standard call-by-value translation of λ calculus.

means that all the effects of the original computation are preserved, because *join* combines the effects of the two computations. All four laws hold for the two implementations of *malias* presented in the previous section. Since the two implementations only use monad operations, we can prove that the laws hold for any monad using just the standard monad laws. The proofs can be found in Appendix B.

We return to the discussion about laws in Section 3, where we introduce the category theoretical background in more details. Before doing that, we give a formal description of the translation algorithm.

2.3 Lambda calculus translation

The *call-by-name* and *call-by-value* translations that we demonstrated in Section 1.1 were first formally introduced by Wadler [24]. In this section, we present a similar formal definition of our translation based on the *malias* operation. For our source language, we use a simple λ calculus with let-binding:

$$
\begin{array}{lllll}
e & \in & Expr & e & ::= & x \mid \lambda x.e \mid e_1\, e_2 \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \\
\tau & \in & Type & \tau & ::= & \alpha \mid \tau_1 \to \tau_2
\end{array}
$$

The target language of the translation is the same, with the only extension – it contains an additional type scheme $M\ \tau$. The *call-by-name* and *call-by-value* translation of the lambda calculus are shown in Figure 2 and Figure 3, respectively. In the translation, we alias $\gg\!=$ as *bind*. The translation of types and typing judgements are omitted for simplicity and can be found in the original paper [24].

Our translation, called *call-by-alias*, is presented below. The translation has similar structure to *call-by-name*, with the exception that it inserts *malias* operation in the last two cases:

$$
\begin{array}{lcl}
[\![\alpha]\!]_{cba} & = & M\ \alpha \\
[\![\tau_1 \to \tau_2]\!]_{cba} & = & M\ ([\![\tau_1]\!]_{cba} \to [\![\tau_2]\!]_{cba}) \\
\\
[\![x]\!]_{cba} & = & x \\
[\![\lambda x.e]\!]_{cba} & = & unit\ (\lambda x.[\![e]\!]_{cba}) \\
[\![e_1\, e_2]\!]_{cba} & = & bind\ [\![e_1]\!]_{cba}\ (\lambda f.bind\ (malias\ [\![e_2]\!]_{cba})\ f) \\
[\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]_{cba} & = & bind\ (malias\ [\![e_1]\!]_{cba})\ (\lambda x.[\![e_2]\!]_{cba})
\end{array}
$$

$$
[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau]\!]_{cba} = x_1 : [\![\tau_1]\!]_{cba}, \ldots, x_n : [\![\tau_n]\!]_{cba} \vdash [\![e]\!]_{cba} : [\![\tau]\!]_{cba}
$$

The translation turns all user-defined variables of type τ into variables of type $M\ \tau$. A variable access expression x is translated to just a variable access, which now represents a computation (that may possibly carry some effects). A lambda expression is turned into a lambda expression wrapped in a pure monadic computation.

The two interesting cases are application and let-binding. When translating function application, we bind on the computation representing the function. We want to call the function f with an aliased computation as an argument. This is achieved by passing the translated argument to *malias* and then applying *bind* again. The translation of let-binding is similar, but slightly simpler, because it does not need to use *bind* to obtain a function.

The listing also shows how typing judgements are translated. This is used to show that the translation preserves typing. A well-typed term in the original language translates to a well-typed term in the target language. The details of the proof can be found in Appendix C.

3 Computational semi-bimonads

In this section, we formally describe the structure that underlies a monad having a *malias* operation as described and used in the previous section. As already mentioned, the *malias* operation corresponds to an operation of an associated comonad, so we first review the definitions of a monad and a comonad. Monads are well-known structures in functional programming. Comonads are dual structures to monads that are less widespread, but they have also been used in functional programming and semantics (Section 5):

Definition 1. *A monad over a category $\mathscr{C}$ is a triple (T, η, μ) where $T : \mathscr{C} \to \mathscr{C}$ is a functor, $\eta : I_{\mathscr{C}} \to T$ is a natural transformation from the identity functor to T, and $\mu : T^2 \to T$ is a natural transformation, such that the following associativity and identity conditions hold, for every object A:*

- $\mu_A \circ T\mu_A = \mu_A \circ \mu_{TA}$
- $\mu_A \circ \eta_{TA} = \mathsf{id}_{TA} = \mu_A \circ T\eta_A$

Definition 2. *A comonad over a category $\mathscr{C}$ is a triple (T, ε, δ) where $T : \mathscr{C} \to \mathscr{C}$ is a functor, $\varepsilon : T \to I_{\mathscr{C}}$ is a natural transformation from T to the identity functor, and $\delta : T \to T^2$ is a natural transformation from T to T^2, such that the following associativity and identity conditions hold, for every object A:*

- $T\delta_A \circ \delta_A = \delta_{TA} \circ \delta_A$
- $\varepsilon_{TA} \circ \delta_A = \mathsf{id}_{TA} = T\varepsilon_A \circ \delta_A$

In functional programming terms, the natural transformation η corresponds to $\mathit{unit} :: a \to m\ a$ and the natural transformation μ corresponds to $\mathit{join} :: m\ (m\ a) \to m\ a$. A comonad is a dual structure to a monad – the natural transformation ε corresponds to an operation $\mathit{counit} :: m\ a \to a$ and δ corresponds to $\mathit{cojoin} :: m\ a \to m\ (m\ a)$. An equivalent formulation of comonads in functional programming uses an operation $\mathit{cobind} :: m\ a \to (m\ a \to b) \to m\ b$, which is dual to $\gg\!=$ of monads.

A simple example of a comonad is the product comonad. The type $m\ a$ stores the value of a and some additional state S, meaning that $TA = A \times S$. The ε (or *counit*) operation extracts the value A ignoring the additional state. The δ (or *cojoin*) operation duplicates the state. In functional programming, the product comonad is equivalent to the reader monad $TA = S \to A$.

In this paper, we use a special variant of comonads. Computational comonads, introduced by Brookes and Geva [5], have an additional operation γ together with laws specifying its properties:

Definition 3. *A computational comonad over a category $\mathscr{C}$ is a quadruple $(T,\varepsilon,\delta,\gamma)$ where (T,ε,δ) is a comonad over $\mathscr{C}$ and $\gamma : I_{\mathscr{C}} \to T$ is a natural transformation such that, for every object A,*

- $\varepsilon_A \circ \gamma_A = \mathsf{id}_A$
- $\delta_A \circ \gamma_A = \gamma_{TA} \circ \gamma_A$.

A computational comonad has an additional operation γ which has the same type as the η operation of a monad, that is $a \to m\ a$. In the work on computational comonads, the transformation γ turns an extensional specification into an intensional specification without additional computational information.

In our work, we do not need the natural transformation corresponding to $counit :: m\ a \to a$. We define a computational *semi*-comonad, which is a computational comonad without the natural transformation ε and without the associated laws. The remaining structure is preserved:

Definition 4. *A computational semi-comonad over a category $\mathscr{C}$ is a triple (T,δ,γ) where $T : \mathscr{C} \to \mathscr{C}$ is a functor, $\delta : T \to T^2$ is a natural transformation from T to T^2 and $\gamma : I_{\mathscr{C}} \to T$ is a natural transformation from the identity functor to T, such that the following associativity and computationality conditions hold, for every object A:*

- $T\delta_A \circ \delta_A = \delta_{TA} \circ \delta_A$
- $\delta_A \circ \gamma_A = \gamma_{TA} \circ \gamma_A$.

Finally, to define a structure that models our monadic computations with the *malias* operation, we need to combine the definition of a *monad* and *computational semi-comonad*. We require that the two structures share the functor T and that the natural transformation $\eta : I_{\mathscr{C}} \to T$ of a monad coincides with the natural transformation $\gamma : I_{\mathscr{C}} \to T$ of a computational comonad.

Definition 5. *A computational semi-bimonad over a category $\mathscr{C}$ is a quadruple (T,η,μ,δ) where (T,η,μ) is a monad over a category $\mathscr{C}$ and (T,δ,η) is a computational semi-comonad over $\mathscr{C}$, such that the following additional condition holds, for every object A:*

- $\mu_A \circ \delta_A = \mathsf{id}_{TA}$

The definition of *computational semi-bimonad* relates the monadic and comonadic parts of the structure using an additional law. Given an object A, the law specifies that taking TA to T^2A using the natural transformation δ_A of a comonad and then back to TA using the natural transformation μ_A is identity.

3.1 Revisiting the laws

The laws of computational semi-bimonad as defined in the previous section are exactly the laws of our monad equipped with the *malias* operation. In this section, we briefly review the laws and present the category theoretic version of all the laws demonstrated in Section 2.2. We require four laws in addition to the standard monad laws. A diagrammatic demonstration is shown in Figure 4. For all objects A and B of $\mathscr{C}$ and for all $f : A \to B$ in $\mathscr{C}$:

$$
\begin{array}{rcll}
T^2 f \circ \delta_A & = & \delta_B \circ Tf & (\textit{naturality}) \\
T\delta_A \circ \delta_A & = & \delta_{TA} \circ \delta_A & (\textit{associativity}) \\
\delta_A \circ \eta_A & = & \eta_{TA} \circ \eta_A & (\textit{computationality}) \\
\mu_A \circ \delta_A & = & \mathsf{id}_{TA} & (\textit{identity})
\end{array}
$$

$$\begin{tikzcd} TA \arrow[r, "Tf"] \arrow[d, "\delta_A"'] & TB \arrow[d, "\delta_B"] \\ T^2A \arrow[r, "T^2f"'] & T^2B \end{tikzcd} \qquad \begin{tikzcd} TA \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] & T^2A \arrow[d, "\delta_{TA}"] \\ T^2A \arrow[r, "T\delta_A"'] & T^3A \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[d, "\eta_A"'] & TA \arrow[d, "\delta_A"] \\ TA \arrow[r, "\eta_{TA}"'] & T^2A \end{tikzcd} \qquad \begin{tikzcd} TA \arrow[rr, "\mathsf{id}_{TA}"] \arrow[dr, "\delta_A"'] & & TA \\ & T^2A \arrow[ur, "\mu_A"'] & \end{tikzcd}$$

(*naturality*) (*associativity*) (*computationality*) (*identity*)

Figure 4: Diagramatic representation of the four additional properties of semi-bimonads

The *naturality* law follows from the fact that δ is a natural transformation and so we did not state it explicitly in Definition 5. However, it is one of the laws that are translated to the functional programming interpretation. The *associativity* law is a law of comonad – the other law in Definition 2 does not apply in our scenario, because we only work with *semi*-comonad that does not have natural transformation ε (*counit*). The *computationality* law is a law of computational comonad and finally, the *identity* law is the additional law of *computational semi-bimonads*.

4 Abstracting evaluation strategy in practice

In this section, we present several practical uses of the *malias* operation. We start by showing how to write monadic code that is parameterized over the evaluation strategy and then look how to implement *call-by-need* in this framework. Then we also briefly consider *parallel call-by-need* and the relation between *malias* and *joinads* and the **docase** notation [19].

4.1 Parameterization by evaluation strategy

One of the motivations of this work is that the standard monadic translations for *call-by-name* and *call-by-value* produce code with different structure. We showed an alternative translation that can be used with both of the evaluation strategies just by changing the definition of the *malias* operation. In this section, we make one more step – we show how to write code parameterized by evaluation strategy.

We define a *monad transformer* [13] that takes a monad and turns it into a monad with *malias* that implements a specific evaluation strategy. Our example can then be implemented using functions that are polymorphic over the monad transformer. We continue using the previous example based on the *IO* monad, but the transformer can operate on any monad.

As a first step, we define a type class named *MonadAlias* that extends *Monad* with the *malias* operation. To keep the code simple, we do not include comments documenting the laws:

class *Monad m* $\Rightarrow$ *MonadAlias m* **where**
  malias :: *m a* $\rightarrow$ *m* (*m a*)

Next, we define two new types that represent monadic computations using the *call-by-name* and *call-by-value* evaluation strategy. The two types are wrappers that make it possible to implement two different instances of *MonadAlias* for any underlying monadic computation *m a*:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow Monad\ (CbV\ m)\ \textbf{where} \\
\quad return\ v = CbV\ (return\ v) \\
\quad (CbV\ a) \gg\!= f = CbV\ (a \gg\!= (runCbV \circ f)) \\
\textbf{instance}\ Monad\ m \Rightarrow Monad\ (CbN\ m)\ \textbf{where} \\
\quad return\ v = CbN\ (return\ v) \\
\quad (CbN\ a) \gg\!= f = CbN\ (a \gg\!= (runCbN \circ f)) \\
\\
\textbf{instance}\ MonadTrans\ CbV\ \textbf{where}\ lift = CbV \\
\textbf{instance}\ MonadTrans\ CbN\ \textbf{where}\ lift = CbN \\
\\
\textbf{instance}\ Monad\ m \Rightarrow MonadAlias\ (CbV\ m)\ \textbf{where} \\
\quad malias\ m = m \gg\!= (return \circ return) \\
\textbf{instance}\ Monad\ m \Rightarrow MonadAlias\ (CbN\ m)\ \textbf{where} \\
\quad malias\ m = return\ m
\end{array}
$$

Figure 5: Instances of *Monad*, *MonadTrans* and *MonadAlias* for evaluation strategies

$$
\begin{array}{l}
\textbf{newtype}\ CbV\ m\ a = CbV\ \{runCbV :: m\ a\} \\
\textbf{newtype}\ CbN\ m\ a = CbN\ \{runCbN :: m\ a\}
\end{array}
$$

The snippet defines types *CbV* and *CbN* that represent two evaluation strategies. Figure 5 shows the implementation of three type classes for these two types. The implementation of the *Monad* type class is the same for both of the types, because it simply uses *return* and $\gg\!=$ operations of the underlying monad. The implementation of *MonadTrans* wraps a monadic computation $m\ a$ into a type $CbV\ m\ a$ and $CbN\ m\ a$, respectively. Finally, the instances of the *MonadAlias* type class associate the two implementations of *malias* (from Section 2.1) with the two data types.

Example. Using the previous definitions, we can now rewrite the example from Section 1.1 using generic functions that can be executed using both *runCbV* and *runCbN*. Instead of implementing *malias* for a specific monad such as $IO\ a$, we use a monad transformer t that lifts the monadic computation to either $CbV\ IO\ a$ or to $CbN\ IO\ a$. This means that all functions will have constraints $MonadTrans\ t$, specifying that t is a monad transformer, and $MonadAlias\ (t\ m)$, specifying that the computation implements the *malias* operation.

In Haskell, this can be succinctly written using *constraint kinds* [3], that make it possible to define a single constraint $EvalStrategy\ t\ m$ that combines both of the conditions[1]:

$$\textbf{type}\ EvalStrategy\ t\ m = (MonadTrans\ t, MonadAlias\ (t\ m))$$

Despite the use of the **type** keyword, the identifier *EvalStrategy* actually has a kind *Constraint*, which means that it can be used to specify assumptions about types in a function signature. In our example, we write $EvalStrategy\ t\ IO$ to specify that a monadic computation uses the *IO* monad and some implementation of *MonadAlias* that will be specified later:

[1] Constraint kinds are available in GHC 7.4 and generalize constraint synonyms proposed by Orchard and Schrijvers [18]

$$
\begin{array}{l}
\mathit{chooseSize} :: \mathit{EvalStrategy}\ t\ \mathit{IO} \Rightarrow t\ \mathit{IO}\ \mathit{Int} \to t\ \mathit{IO}\ \mathit{Int} \to t\ \mathit{IO}\ \mathit{Int} \\
\mathit{chooseSize}\ \mathit{new}\ \mathit{legacy} = \mathbf{do} \\
\quad \mathit{newVal} \leftarrow \mathit{new} \\
\quad \mathbf{if}\ \mathit{newVal} > 0\ \mathbf{then}\ \mathit{new}\ \mathbf{else}\ \mathit{legacy} \\
\mathit{fileSize} :: \mathit{EvalStrategy}\ t\ \mathit{IO} \Rightarrow t\ \mathit{IO}\ \mathit{Int} \\
\mathit{fileSize} = \mathbf{do} \\
\quad \mathit{new} \leftarrow \mathit{malias}\ \$\ \mathit{lift}\ (\mathit{lookupInput}\ \texttt{"new\_size"}) \\
\quad \mathit{legacy} \leftarrow \mathit{malias}\ \$\ \mathit{lift}\ (\mathit{lookupInput}\ \texttt{"legacy\_size"}) \\
\quad \mathit{chooseSize}\ \mathit{new}\ \mathit{legacy}
\end{array}
$$

Compared to the previous version of the example, the only significant change is in the type signature of the two functions. Instead of passing computations of type *IO a*, they now work with computations *t IO a* with the constraint *EvalStrategy t IO*. A minor change is also needed, because result of *lookupInput* is still *IO Int*. To lift it into a monadic computation with evaluation strategy, we use the *lift* function provided by a monad transformer.

The return type of the *fileSize* computation is parameterized over the evaluation strategy *t*. This means that we can call it in two different ways. Writing *runCbN fileSize* executes the function using *call-by-name* and writing *runCbV fileSize* executes it using *call-by-value*. However, it is also possible to implement a monad transformer for *call-by-need*.

4.2 Implementing call-by-need strategy

In this section, we answer an open question posed by Wadler when introducing the use of monads in functional programming [24]. We show how to use the mechanisms described so far to give a *call-by-need* translation of purely functional programs.

In the absence of effects, the *call-by-need* strategy is equivalent to the *call-by-name* strategy, with the only difference that performance characteristics may differ. In the *call-by-need* (or *lazy*)strategy, a computation passed as an argument is evaluated at most once and the result is cached afterwards.

The caching of results needs to be done in the *malias* operation. This cannot be done for *any* monad, but we can define a monad transformer similar to the ones presented in the previous section. In particular, we use a monad transformer [13] based on the *ST* monad [11]:

$$\mathbf{newtype}\ \mathit{CbL}\ s\ m\ a = \mathit{CbL}\ \{\mathit{unCbL} :: \mathit{STT}\ s\ m\ a\}$$

Unlike *CbV* and *CbN*, the *CbL* type is not a simple wrapper that contains a computation of type *m a*. Instead, it contains a computation augmented with some additional state. The state is used for caching the values of evaluated computations. The type *STT s m a* represents a computation *m a* with an additional local state tagged with a type variable *s*. The use of a local state instead of e.g. *IO* means that the monadic computation can be safely evaluated even as part of purely functional code. The tags are used merely to guarantee that state associated with one *STT* monad does not leak to other parts of the program.

Implementing the *Monad* and *MonadTrans* instances follows exactly the same pattern as instances for other transformers in Figure 5. The interesting work is done in the *malias* function of *MonadAlias*:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadAlias}\ (\mathit{CbL}\ s\ m)\ \mathbf{where} \\
\quad \mathit{malias}\ (\mathit{CbL}\ \mathit{marg}) = \mathit{CbL}\ \$\ \mathbf{do} \\
\quad\quad r \leftarrow \mathit{newSTRef}\ \mathit{Nothing} \\
\quad\quad \mathit{return}\ (\mathit{CbL}\ \$\ \mathbf{do}
\end{array}
$$

$$\begin{array}{l}
rv \leftarrow readSTRef\ r \\
\textbf{case}\ rv\ \textbf{of} \\
\quad Nothing \rightarrow marg \gg\!= \lambda v \rightarrow writeSTRef\ r\ (Just\ v) \gg return\ v \\
\quad Just\ v \rightarrow return\ v)
\end{array}$$

The *malias* operation takes a computation of type $m\ a$ and returns a computation $m\ (m\ a)$. By wrapping the underlying monad m using a monad transformer, this essentially means that the result should be a computation $STT\ s\ m\ (STT\ s\ m\ a)$, wrapped in the *CbL* constructor.

Both the outer and the nested computations are augmented using a state transformer that makes it possible to access a shared mutable state. This means that the outer computation can allocate a new reference and the inner computation can use it to access and store the result computed previously. The allocation is done using the *newSTRef* function, which creates a reference initialized to *Nothing*. In the returned (inner) computation, we first read the state using *readSTRef*. If the value was computed previously, then it is simply returned. If not, the computation evaluates *marg*, stores the result in a reference cell and then returns the obtained value.

After implementing *runCbL* function (which can be done easily using *runST*), we can use the type *CbL* to execute the example from Section 4.1. Using the *call-by-need* semantics, the program finally behaves in a desirable way. If the value of the `"new_size"` key is greater than zero, then it reads it only once, without reading the value of the `"legacy_size"` key. The value of `"legacy_size"` key is accessed only if the value of the `"new_size"` key is less than zero.

4.3 Parallel call-by-need strategy

In this section, we consider yet another evaluation strategy that can be implemented using our scheme. The *parallel call-by-need* strategy [2] is similar to *call-by-need*, but it may optimistically start evaluating a computation sooner, in parallel with the main program. When carefully tuned, the evaluation strategy may result in a better performance on multi-core CPUs.

We present a simple implementation of the *malias* operation based on the monad for deterministic parallelism by Marlow et al. [14]. By translating purely functional code to a monadic version using our translation and the *Par* monad, we get a program that attempts to evaluate arguments of every function call in parallel. In practice, this may introduce too much overhead, but it demonstrate that *parallel call-by-need* strategy also fits with our general framework.

Unlike the previous two sections, we do not define a monad transformer that can embody any monadic computation. For example, performing IO operations in parallel might introduce non-determinism. Instead, we implement *malias* operation directly for computations of type *Par a*:

$$\begin{array}{l}
\textbf{instance}\ MonadAlias\ Par\ \textbf{where} \\
\quad malias\ m = \\
\quad\quad spawn\ m \gg\!= return \circ get
\end{array}$$

The implementation is surprisingly simple. The function *spawn* creates a computation $Par\ (IVar\ a)$ that starts the work in background and returns a mutable variable (I-structure) that will contain the result when the computation completes. The inner computation that is returned by *malias* calls a function $get :: IVar\ a \rightarrow Par\ a$ that waits until *IVar* has been assigned a value and then returns it.

Using the above implementation of *malias*, we can now translate purely functional code to a monadic version that uses *parallel call-by-need* instead of the previous standard evaluation strategies (that do not introduce any parallelism). For example, consider a naive Fibonacci function:

$$\begin{aligned}&\mathit{fibSeq}\ n \mid n \leqslant 1 = n\\&\mathit{fibSeq}\ n = \mathit{fibSeq}\ (n-1) + \mathit{fibSeq}\ (n-2)\end{aligned}$$

The second case calls the $+$ operator with two computations as arguments. The translated version passes these computations to *malias*, which starts executing them in parallel. The monadic version of the $+$ operator then waits until both computations complete. If we translate only the second case and manually add a case that calls sequential version of the function for inputs smaller than 30, we get the following code:

$$\begin{aligned}&\mathit{fibPar}\ n \mid n < 30 = \mathit{return}\ (\mathit{fibSeq}\ n)\\&\mathit{fibPar}\ n = \mathbf{do}\\&\quad \mathit{n1} \leftarrow \mathit{malias}\,\$\,\mathit{fibPar}\ (n-1)\\&\quad \mathit{n2} \leftarrow \mathit{malias}\,\$\,\mathit{fibPar}\ (n-2)\\&\quad \mathit{liftM2}\ (+)\ \mathit{n1}\ \mathit{n2}\end{aligned}$$

Aside from the first line, the code directly follows the general translation mechanism described earlier. Arguments of a function are turned to monadic computations and passed to *malias*. The inner computations obtained using monadic bind are then passed to a translated function.

Thanks to the manually added optimization that calls *fibSeq* for smaller inputs, the function runs nearly twice as fast on a dual-core machine[2]. The use of *malias* for parallel programming is one of the first areas where we found the operation useful. We first considered it as part of *joinads*, which are discussed in the next section.

4.4 Simplifying Joinads

Joinads [20, 19] were designed to simplify programming with certain kinds of monadic computations. Many monads, especially from the area of concurrent or parallel programming provide additional operations for composing monadic computations – in particular, *parallel composition*, *(non-deterministic) choice* and *aliasing*. However, these operations were added in an ad-hoc fashion, which means that developers need to understand each monadic library separately and there was no unifying syntax.

Joinads are abstract computations that form a monad and provide the three additional operations mentioned above. The work on joinads also introduces a syntactic extension for Haskell and F# that makes it easier to work with these classes of computations. For example, the following snippet uses the *Par* monad to implement a function that tests whether a predicate holds for all leafs of a tree in parallel:

$$\begin{aligned}&\mathit{all} :: (a \rightarrow \mathit{Bool}) \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Par}\ \mathit{Bool}\\&\mathit{all}\ p\ (\mathit{Leaf}\ v) \qquad\qquad = \mathit{return}\ (p\ v)\\&\mathit{all}\ p\ (\mathit{Node}\ \mathit{left}\ \mathit{right}) =\\&\quad \mathbf{docase}\ (\mathit{all}\ p\ \mathit{left}, \mathit{all}\ p\ \mathit{right})\ \mathbf{of}\\&\qquad (\mathit{False}, ?) \quad \rightarrow \mathit{return}\ \mathit{False}\\&\qquad (?, \mathit{False}) \quad \rightarrow \mathit{return}\ \mathit{False}\\&\qquad (\mathit{allL}, \mathit{allR}) \rightarrow \mathit{return}\ (\mathit{allL} \wedge \mathit{allR})\end{aligned}$$

The **docase** notation intentionally resembles pattern matching. It has similar semantics as well. The first two cases use the special pattern ? to denote that the value of one of the computations does not have to be available in order to continue. When one of the sub-branches returns *False*, we know that the overall

[2] When called with input 37 on a Core 2 Duo CPU, the sequential version runs in 8.9s and the parallel version in 5.1s.

result is *False* and so we return immediately. Finally, the last clause matches if none of the two previous does. It can only match after both sub-trees are processed.

Similarly to the **do** notation, the **docase** syntax is desugared to uses of the joinad operations. The choice between clauses is translated using the *choice* operator. If a clause requires the result of multiple computations (such as the last one), the computations are combined using *parallel composition*.

If a computation, passed as an argument, is accessed from multiple clauses, then it should be evaluated only once and the clauses should only access *aliased* computation. This motivation is similar to the one described in this article. Indeed, joinads use a variant of the *malias* operation and insert it automatically for all arguments of **docase**. This is very similar to how the translation presented in this paper uses *malias*. If aliasing was not done automatically behind the scenes, we would have to write:

$$
\begin{array}{l}
\mathit{all}\ p\ (\mathit{Leaf}\ v) \qquad\qquad = \mathit{return}\ (p\ v) \\
\mathit{all}\ p\ (\mathit{Node}\ \mathit{left}\ \mathit{right}) = \mathbf{do} \\
\quad l \leftarrow \mathit{malias}\ (\mathit{all}\ p\ \mathit{left}) \\
\quad r \leftarrow \mathit{malias}\ (\mathit{all}\ p\ \mathit{right}) \\
\quad \mathbf{docase}\ (l, r)\ \mathbf{of} \\
\qquad (\mathit{False}, ?) \quad \rightarrow \mathit{return}\ \mathit{False} \\
\qquad (?, \mathit{False}) \quad \rightarrow \mathit{return}\ \mathit{False} \\
\qquad (\mathit{allL}, \mathit{allR}) \rightarrow \mathit{return}\ (\mathit{allL} \wedge \mathit{allR})
\end{array}
$$

One of the limitations of the original design of joinads is that there is no one-to-one correspondence between the **docase** notation and what can be expressed directly using the joinad operations. This is partly due to the automatic aliasing of arguments, which inserts *malias* only at very specific locations. We believe that integrating a *call-by-alias* translation, described in this article, in a programming language could resolve this situation. It would also separate the two concerns – composition of computations using *choice* and *parallel composition* (done by joinads) and automatic aliasing of computations that allows sharing of results as in *call-by-need* or *parallel call-by-need*.

5 Related work

Most of the work that directly influenced this work has been discussed throughout the paper. Most importantly, the question of translating pure code to a monadic version with *call-by-need* semantics was posed by Wadler [24]. To our knowledge, this question has not been answered before, but there is various work that either uses similar structures or considers evaluation strategies from different perspectives.

Monads with aliasing. There are numerous monads that independently introduced an operation of type $m\ a \rightarrow m\ (m\ a)$. The *eagerly* combinator from the Orc monad [10] causes computation to run in parallel, effectively implementing the *parallel call-by-need* evaluation strategy. The only law that relates *eagerly* to operations of a monad is too restrictive to allow the *call-by-value* semantics.

A monad for purely-functional lazy non-deterministic programming [6] uses a similar combinator *share* to make monadic (non-deterministic) computations lazy. However, *call-by-value* strategy is inefficient and the *call-by-name* strategy is incorrect, because choosing a different value each time a non-deterministic computation is accessed means that *generate and test* pattern does not work.

The *share* operation is described together with the laws that should hold. The *HNF* law is similar to our *computationality*. However, the *Ignore* law specifies that the *share* operation should not be strict (ruling out our *call-by-value* implementation of *malias*). A related paper [4] discusses where *share* needs to be inserted when translating lazy non-deterministic programs to a monadic form. The results may be directly applicable to make our translation more efficient by inserting *malias* only when required.

Abstract computations and comonads. In this paper, we extended monads with a part of comonadic structure. Although less widespread than monads, comonads are also useful to capture abstract computations in functional programming. They have been used for dataflow programming [23], array programming [17], environment passing, and more [8]. In general, comonads can be used to describe context-dependent computations [22], where *cojoin* (natural transformation δ) duplicates the context. In our work, the corresponding operation *malias* splits the context (effects) in a particular way between two computations.

We only considered basic monadic computations, but it would be interesting to see how *malias* interacts with other abstract notions of computations, such as *applicative functors* [15], *arrows* [7] or additive monads (the *MonadPlus* type-class). The monad for lazy non-deterministic programming [6], mentioned earlier, implements *MonadPlus* and may thus provide interesting insights.

Evaluation strategies. One of the key results of this paper is that it gives a monadic translation from purely functional code to a monadic version that has the *call-by-need* semantics. We achieve that using the monad transformer [13] for adding state.

The semantics of *call-by-need* is equivalent to *call-by-name*, but it has been described formally as a version of λ-calculus by Ariola and Felleisen [1]. This allows equational reasoning about computations and it could be used to show that our encoding directly corresponds to *call-by-need*, similarly to proofs for other strategies in Appendix A. The semantics has been also described using an environment that models the caching [9, 21], which closely corresponds to our actual implementation.

Considering the two basic evaluation strategies, Wadler [25] shows that *call-by-name* is dual to *call-by-value*. We find this curious as the two definitions of *malias* in our work are, in some sense, also dual or symmetric as they associate all effects with the inner or the outer monad of the type $m\ (m\ a)$. Furthermore, we believe that a logical interpretation of our generalized evaluation strategy may be an interesting future work.

Finally, the work presented in this work unifies monadic *call-by-need* and *call-by-value*. In the non-monadic setting, a similar goal is achieved by the *call-by-push-value* calculus [12]. The calculus is more fine-grained and strictly separates *values* and *computations*. Using these mechanisms, it is possible to encode both *call-by-need* and *call-by-value*. It may be interesting to consider whether our computations parameterized over evaluation strategy (Section 4.1) could be encoded in *call-by-push-value*.

6 Conclusions

In this paper we presented an alternative translation from purely functional code to monadic form. Previously, this required choosing either *call-by-need* or *call-by-value* translation and the translated code had different structure and different types in both cases. Our translation abstracts the evaluation strategy into a function *malias* that can be implemented separately to get code with the required evaluation strategy.

Possibly, the main contribution of our work is that the translation can be also used to define monadic computations with the *call-by-need* strategy. This was highlighted as an interesting open problem by Wadler [24]. Our approach has other interesting applications – it makes it possible to write code that is parameterized by the evaluation strategy and it allows implementing *parallel call-by-need* strategy for certain monads.

Finally, we presented the theoretical foundations of our approach using a model described in terms of category theory. We extended monad with an additional operation based on *computational comonads*, which were previously used to give intensional semantics of computations. In our setting, the operation specifies the evaluation order. The categorical model specifies laws about *malias* and we proved that the laws hold for *call-by-value* and *call-by-name* strategies.

Acknowledgements. The author is grateful to Sebastian Fischer, Dominic Orchard and Alan Mycroft for inspiring comments and discussion, and to the latter two for proofreading of the paper. The work has been partly supported by EPSRC and through the Cambridge CHESS scheme.

References

[1] Zena M. Ariola & Matthias Felleisen (1997): *The call-by-need lambda calculus*. *J. Funct. Program.* 7, pp. 265–301, doi:10.1017/S0956796897002724.

[2] Clem Baker-Finch, David J. King & Phil Trinder (2000): *An operational semantics for parallel lazy evaluation*. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, doi:http://doi.acm.org/10.1145/351240.351256.

[3] Max Bolingbroke (2011): *Constraint Kinds for GHC*. Available at `http://blog.omega-prime.co.uk/?p=127`.

[4] B. Braßel, S. Fischer, M. Hanus & F. Reck (2011): *Transforming Functional Logic Programs into Monadic Functional Programs*. In: *Proc. of the 19th International Workshop on Functional and (Constraint) Logic Programming 2010*, WFLP '10, Springer LNCS 6559, pp. 30–47.

[5] Stephen Brookes & Shai Geva (1992): *Computational Comonads and Intensional Semantics*. In: *Applications of Categories in Computer Science: Proceedings LMS Symp., Durham, UK, 20–30 July 1991*, 177, pp. 1–44. Available at `http://citeseer.ist.psu.edu/brookes91computational.html`.

[6] Sebastian Fischer, Oleg Kiselyov & Chung-chieh Shan (2009): *Purely functional lazy non-deterministic programming*. In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pp. 11–22, doi:http://doi.acm.org/10.1145/1596550.1596556.

[7] John Hughes (1998): *Generalising Monads to Arrows*. *Science of Computer Programming* 37, pp. 67–111.

[8] Richard B. Kieburtz (1999): *Codata and Comonads in Haskell*.

[9] John Launchbury (1993): *A natural semantics for lazy evaluation*. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pp. 144–154, doi:http://doi.acm.org/10.1145/158511.158618.

[10] John Launchbury & Trevor Elliott (2010): *Concurrent orchestration in Haskell*. In: *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pp. 79–90, doi:http://doi.acm.org/10.1145/1863523.1863534.

[11] John Launchbury & Simon Peyton Jones: *State in Haskell*. In: *Proceedings of the Lisp and Symbolic Computation conference*, LISP '95.

[12] Paul Blain Levy (2004): *Call-By-Push-Value*. Springer.

[13] Sheng Liang, Paul Hudak & Mark Jones (1995): *Monad transformers and modular interpreters*. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, doi:http://doi.acm.org/10.1145/199448.199528.

[14] Simon Marlow, Ryan Newton & Simon Peyton Jones (2011): *A monad for deterministic parallelism*. In: *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, doi:http://doi.acm.org/10.1145/2034675.2034685.

[15] Conor Mcbride & Ross Paterson (2008): *Applicative programming with effects*. *J. Funct. Program.* 18.

[16] Eugenio Moggi (1991): *Notions of computation and monads*. *Inf. Comput.* 93, pp. 55–92.

[17] Dominic Orchard, Max Bolingbroke & Alan Mycroft (2010): *Ypnos: Declarative, Parallel Structured Grid Programming*. In: *DAMP '10*, ACM, NY, USA, pp. 15–24, doi:http://doi.acm.org/10.1145/1708046.1708053.

[18] Dominic A. Orchard & Tom Schrijvers (2010): *Haskell Type Constraints Unleashed*. In: *FLOPS*, pp. 56–71. Available at `http://dx.doi.org/10.1007/978-3-642-12251-4_6`.

[19] Tomas Petricek, Alan Mycroft & Don Syme: *Extending Monads with Pattern Matching*. In: *Proceedings of Haskell Symposium*, Haskell 2011.

[20] Tomas Petricek & Don Syme (2011): *Joinads: A Retargetable Control-Flow Construct for Reactive, Parallel and Concurrent Programming*. In: *PADL'11*, pp. 205–219.

[21] S. Purushothaman & Jill Seaman (1992): *An Adequate Operational Semantics for Sharing in Lazy Evaluation*. In: *Proceedings of the 4th European Symposium on Programming*, pp. 435–450.

[22] Tarmo Uustalu & Varmo Vene (2008): *Comonadic Notions of Computation*. *Electron. Notes Theor. Comput. Sci.* 203, pp. 263–284.

[23] Tarmo Uustalu & Varmo Vene (Nov 2006): *The Essence of Dataflow Programming*. *Lecture Notes in Computer Science* 4164, pp. 135–167.

[24] Philip Wadler (1990): *Comprehending monads*. In: *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pp. 61–78.

[25] Philip Wadler (2003): *Call-by-value is dual to call-by-name*. In: *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pp. 189–201, doi:http://doi.acm.org/10.1145/944705.944723.

$$
\begin{array}{lll}
\multicolumn{2}{l}{[\![e_1\, e_2]\!]_{cba}} & \\
\quad = & bind\ [\![e_1]\!]_{cba}\ (\lambda f.bind\ (malias\ [\![e_2]\!]_{cba})\ f)) & \text{(translation)} \\
\quad = & bind\ [\![e_1]\!]_{cba}\ (\lambda f.bind\ (unit\ [\![e_2]\!]_{cba})\ f)) & \text{(definition)} \\
\quad = & bind\ [\![e_1]\!]_{cbn}\ (\lambda f.f\ [\![e_2]\!]_{cbn}) & \text{(left identity)} \\
\quad = & [\![e_1\, e_2]\!]_{cbn} & \text{(translation)}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{[\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]_{cba}} & \\
\quad = & bind\ (malias\ [\![e_1]\!]_{cba})\ (\lambda x.[\![e_2]\!]_{cba}) & \text{(translation)} \\
\quad = & bind\ (unit\ [\![e_1]\!]_{cba})\ (\lambda x.[\![e_2]\!]_{cba}) & \text{(definition)} \\
\quad = & (\lambda x.[\![e_2]\!]_{cbn})\ [\![e_1]\!]_{cbn} & \text{(left identity)} \\
\quad = & [\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]_{cbn} & \text{(translation)}
\end{array}
$$

Figure 6: Proving that using an appropriate *malias* is equivalent to *call-by-name*.

A Equivalence proofs

In this section, we show that our translation presented in Section 2.3 can be used to implement standard *call-by-name* and *call-by-value*. We prove that using an appropriate definition of *malias* from Section 2.1 gives the same semantics as the standard translations described by Wadler [24].

Call-by-name. The translation of types is the same for our translation and the *call-by-name* translation. In addition, the rules for translating variable access and lambda functions are also the same. This means that we only need to prove that our translation of let-binding and function application are equivalent. When implementing *call-by-name* using our translation, we use the following definition of *malias*:

$$malias\ m = unit\ m$$

Using this definition and the *left identity* monad law, we can now show that our *call-by-alias* translation is equivalent to the translation from Figure 2. The Figure 6 shows the equations for function application and let-binding.

Call-by-value. Proving that appropriate definition of *malias* gives a term that corresponds to the one obtained using standard *call-by-value* translation is more difficult. In the *call-by-value* translation, functions are translated to a type $\tau_1 \to M\ \tau_2$, while our translation produces functions of type $M\ \tau_1 \to M\ \tau_2$. As a reminder, the definition of *malias* that gives the *call-by-value* behaviour is the following:

$$malias\ m = bind\ m\ (unit \circ unit)$$

To prove that the two translations are equivalent, we show that the following invariant holds: When using the above definition of *malias* and our *call-by-alias* translation, the monadic computations of type $M\ \tau$ that are assigned to variables always have a structure $unit\ x_v$ where x_v is a variable of type τ.

Using this invariant, we can show that a *call-by-value* variable access that is translated as $unit\ x$ is equivalent to *call-by-alias* variable access translated as x where the value given to x is $unit\ x_v$. Next, we need to show that a value assigned to a variable in let-binding and function application has the expected structure. The proof details are shown in Figure 7.

$$
\begin{aligned}
& [\![x]\!]_{cba} \\
& \quad = x && \text{(translation)} \\
& \quad = \mathit{unit}\ x_v && \text{(invariant)} \\
& \quad = [\![x_v]\!]_{cbv} && \text{(translation)}
\end{aligned}
$$

$$
\begin{aligned}
& [\![e_1\ e_2]\!]_{cba} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cba}\ (\lambda f.\mathit{bind}\ (\mathit{malias}\ [\![e_2]\!]_{cba})\ f)) && \text{(translation)} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cba}\ (\lambda f.\mathit{bind}\ (\mathit{bind}\ [\![e_2]\!]_{cba}\ (\mathit{unit} \circ \mathit{unit}))\ f)) && \text{(definition)} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cba}\ (\lambda f.\mathit{bind}\ [\![e_2]\!]_{cba}\ (\lambda x_v.\mathit{bind}\ (\mathit{unit}\ (\mathit{unit}\ x_v))\ f) && \text{(associativity)} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cba}\ (\lambda f.\mathit{bind}\ [\![e_2]\!]_{cba}\ (\lambda x_v.f\ (\mathit{unit}\ x_v)) && \text{(left identity)} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cbv}\ (\lambda f.\mathit{bind}\ [\![e_2]\!]_{cbv}\ (\lambda x.f\ x)) && \text{(invariant)} \\
& \quad = [\![e_1\ e_2]\!]_{cbv} && \text{(translation)}
\end{aligned}
$$

$$
\begin{aligned}
& [\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]_{cba} \\
& \quad = \mathit{bind}\ (\mathit{malias}\ [\![e_1]\!]_{cba})\ (\lambda x.[\![e_2]\!]_{cba}) && \text{(translation)} \\
& \quad = \mathit{bind}\ (\mathit{bind}\ [\![e_1]\!]_{cba}\ (\mathit{unit} \circ \mathit{unit}))\ (\lambda x.[\![e_2]\!]_{cba}) && \text{(definition)} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cba}\ (\lambda x_v.\mathit{bind}\ (\mathit{unit}\ (\mathit{unit}\ x_v))\ (\lambda x.[\![e_2]\!]_{cba})) && \text{(associativity)} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cba}\ (\lambda x_v.(\lambda x.[\![e_2]\!]_{cba})\ (\mathit{unit}\ x_v)) && \text{(left identity)} \\
& \quad = \mathit{bind}\ [\![e_1]\!]_{cbv}\ (\lambda x.[\![e_2]\!]_{cbv}) && \text{(invariant)} \\
& \quad = [\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]_{cbv} && \text{(translation)}
\end{aligned}
$$

Figure 7: Proving that using an appropriate *malias* is equivalent to *call-by-value*.

B Proofs for two implementations

In this section, we prove that the two implementations of *malias* presented in Section 2.1 obey the *malias* laws. We use the formulation of monads based on a functor with additional operations *join* and *unit*. The proof relies on the following laws that hold about *join* and *unit*:

$$
\begin{aligned}
\mathit{map}\ (g \circ f) &= (\mathit{map}\ g) \circ (\mathit{map}\ f) && \text{(functor)} \\
\mathit{unit} \circ f &= \mathit{map}\ f \circ \mathit{unit} && \text{(natural unit)} \\
\mathit{join} \circ \mathit{map}\ (\mathit{map}\ f) &= \mathit{map}\ f \circ \mathit{join} && \text{(natural join)} \\
\mathit{join} \circ \mathit{map}\ \mathit{join} &= \mathit{join} \circ \mathit{join} && \text{(assoc join)} \\
\mathit{join} \circ \mathit{unit} &= \mathit{join} \circ \mathit{mapunit} = \mathit{id} && \text{(identity)}
\end{aligned}
$$

The first law follows from the fact that *map* corresponds to a functor. The next two laws hold because *unit* and *join* are both natural transformations. Finally, the last two laws are additional laws that are required to hold about monads (a precise definition can be found in Section 3).

The proofs that the two definitions of *malias* that implement *call-by-name* and *call-by-value* strategies can be found in Figure 8 and Figure 9, respectively. The proofs use the above monad laws and the definitions of *malias* shown in Appendix A. They include proofs for the four additional *malias* laws as defined in Section 2.2: *naturality*, *associativity*, *computationality* and *identity*.

$$
\begin{array}{lll}
\multicolumn{2}{l}{map\ (map\ f) \circ malias} & \\
& = map\ (map\ f) \circ unit & \text{(definition)} \\
& = unit \circ map\ f & \text{(naturality)} \\
& = malias \circ map\ f & \text{(definition)} \\
\\
\multicolumn{2}{l}{map\ malias \circ malias} & \\
& = map\ unit \circ unit & \text{(definition)} \\
& = unit\ \circ\ unit & \text{(natural unit)} \\
& = malias \circ malias & \text{(definition)} \\
\\
\multicolumn{2}{l}{malias \circ unit} & \\
& = unit \circ unit & \text{(definition)} \\
\\
\multicolumn{2}{l}{join \circ malias} & \\
& = join \circ unit & \text{(definition)} \\
& = id & \text{(identity)}
\end{array}
$$

Figure 8: Call-by-name definition of *malias* obeys the laws

$$
\begin{array}{lll}
\multicolumn{2}{l}{map\ (map\ f) \circ malias} & \\
& = map\ (map\ f) \circ map\ unit & \text{(definition)} \\
& = map\ (map\ f \circ unit) & \text{(functor)} \\
& = map\ (unit \circ f) & \text{(natural unit)} \\
& = map\ unit \circ map\ f & \text{(functor)} \\
& = malias\ \circ\ map\ f & \text{(definition)} \\
\\
\multicolumn{2}{l}{map\ malias \circ malias} & \\
& = map\ (map\ unit) \circ (map\ unit) & \text{(definition)} \\
& = map\ (map\ unit \circ unit) & \text{(functor)} \\
& = map\ (unit \circ unit) & \text{(natural unit)} \\
& = map\ unit \circ map\ unit & \text{(functor)} \\
& = malias \circ malias & \text{(definition)} \\
\\
\multicolumn{2}{l}{malias \circ unit} & \\
& = map\ unit \circ unit & \text{(definition)} \\
& = unit \circ unit & \text{(natural unit)} \\
\\
\multicolumn{2}{l}{join \circ malias} & \\
& = join \circ map\ unit & \text{(definition)} \\
& = id & \text{(identity)}
\end{array}
$$

Figure 9: Call-by-value definition of *malias* obeys the laws

C Typing preservation proof

In this section, we show that our translation preserves typing. Given a well-typed term e of type τ, the translated term $[\![e]\!]_{cba}$ is also well-typed and has a type $[\![\tau]\!]_{cba}$. In the rest of this section, we write $[\![-]\!]$ for $[\![-]\!]_{cba}$. To show that the property holds, we use induction over the typing rules, using the fact that $[\![\tau]\!] = M\ \tau'$ for some τ'. The inductive construction of the typing derivation follows the following rules:

$$\text{(var)}\frac{}{[\![\Gamma, x : \tau \vdash x : \tau]\!]} \quad = \quad \frac{}{[\![\Gamma]\!], x : [\![\tau]\!] \vdash x : [\![\tau]\!]}$$

$$\text{(fun)}\frac{[\![\Gamma, x : \tau_1 \vdash e : \tau_2]\!]}{[\![\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2]\!]} \quad = \quad \frac{[\![\Gamma]\!], x : [\![\tau_1]\!] \vdash [\![e]\!] : [\![\tau_2]\!]}{[\![\Gamma]\!] \vdash \mathit{return}\ (\lambda x.[\![e]\!]) : M\ ([\![\tau_1]\!] \to [\![\tau_2]\!])}$$

$$\text{(app)}\frac{[\![\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1]\!]}{[\![\Gamma \vdash e_1\ e_2 : \tau_2]\!]} \quad = \quad \frac{[\![\Gamma]\!] \vdash [\![e_1]\!] : M\ ([\![\tau_1]\!] \to [\![\tau_2]\!]) \quad \Gamma \vdash [\![e_2]\!] : [\![\tau_1]\!]}{\Gamma \vdash \mathit{bind}\ [\![e_1]\!]\ (\lambda f.\mathit{bind}\ (\mathit{malias}\ [\![e_2]\!])\ f)) : [\![\tau_2]\!]}$$

$$\text{(let)}\frac{[\![\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2]\!]}{[\![\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2]\!]} \quad = \quad \frac{[\![\Gamma]\!] \vdash [\![e_1]\!] : [\![\tau_1]\!] \quad [\![\Gamma]\!], x : [\![\tau_1]\!] \vdash [\![e_2]\!] : [\![\tau_2]\!]}{[\![\Gamma]\!] \vdash \mathit{bind}\ (\mathit{malias}\ [\![e_1]\!])\ (\lambda x.[\![e_2]\!]) : [\![\tau_2]\!]}$$