First year report
Effect and coeffect type systems

Tomas Petricek
tomas.petricek@cl.cam.ac.uk

Computer Laboratory,
University of Cambridge

November 24, 2011

Chapter 1

Introduction

In functional languages, especially in the ML-family, types have been traditionally used to track information about data structures. Types classify expressions based on *what* values they compute. However, types can be also used to classify expressions based on *how* they compute values.

We argue that the use of types for the tracking *how* programs compute values is becoming increasingly important. In concurrent programming, an expression may be evaluated on different threads and each thread may have different properties (i.e. allow accessing user-interface or be able to run in the background for a long time), thus it is important to track on which thread can an expression evaluate. In distributed programming (or cloud computing), different parts of a program may evaluate on different nodes (i.e. server, phone or browser), which provide different resources. Code running on an end-user device may execute in a sandbox that provides capabilities to perform only certain operations (i.e. limit access to local storage).

The tracking of *how* programs compute was pioneered by the effect systems introduced by Gifford [12]. Such systems annotate function type with additional information about the operations (such as memory accesses) performed while evaluating the function. Although many effect systems have been designed, their definitions are generally single-purpose, which makes it difficult to support them in a general purpose language.

This report outlines one possible approach to creating a unified framework that could be used to track various kinds of properties of a computation in an ML-style programming language. Chapter 2 describes the work done so far. Aside from reviewing relevant literature, it also introduces *coeffects*, which is a concept dual to *effects* and can be used to describe context-dependent properties of a computation. The Chapter 3 contains a thesis proposal that sketches the future work towards our goal. It describes how a general-purpose framework for tracking of effects and coeffects might look and it also relates the work to the trend of dependently-typed functional programming.

Chapter 2

First year report

This chapter describes the problem and the work done some far. It gives a detailed example of properties that we might want to track in a type system (Section 2.1). Then it reviews relevant literature (Section 2.2) and briefly introduces two ideas that were developed by the author. Section 2.3 introduces the concept of *coeffects* and Section 2.4 introduces *joinads*, an abstract notion of computation that extends monads[1]. Although originally designed for a different purpose, joinads may be also relevant from the type-system point of view (Section 3.3.3).

2.1 Motivation

In the introduction, we argue that tracking how expressions evaluate is becoming increasingly important in modern programming languages. For example, consider the sample program in Figure 2.1. The code is simplified, but realistic example of a client/server program consisting of two functions (see [33] for a more realistic example in F#).

The *dictionaryLookup* function finds a specified word in a dictionary. It uses **access** construct to access a resource representing a database that is only available on the server. The *updateGui* function represents client-side code that reads input entered by the user (by accessing the *getInput* resource). Then it uses **switchTo** keyword to transfer the control flow to a background thread (to avoid blocking the user-interface while waiting for a reply from the server). Next, the function uses **remote** construct to perform a remote procedure call and it switches back to the user-interface thread returns the result.

[1]The work on *joinads* was started during an internship at Microsoft Research, but has been substantially extended during the PhD. It was published and presented at Haskell Symposium 2011 [34]. The work on *coeffects* has been done during the first year, in collaboration with Dominic Orchard and has been submitted to ESOP 2012 [35].

```
let dictionaryLookup = λword →
  let lookupFunc = access dictionaryDatabase
  lookupFunc word

let updateGui = λ() →
  let word = access getInput
  switchTo backgroundThread
  let word = remote dictionaryLookup word
  switchTo guiThread
  translation
```

Figure 2.1: Sample program consisting of a server-side function for dictionary lookup and client-side function for updating the user-interface.

The sample program could be wrong in a number of ways:

- The resource *dictionaryDatabase* is defined only on the server-side and so the function *dictionaryLookup* can be only called on the server. Calling it directly from the client-side code (*updateGui*) would result in an error.
- The remote procedure call using **remote** can take a long time as it involves communication with the server. If the operation is executed on the main user-interface thread, then the end-user application may become unresponsive.
- However, user-interface elements may only be accessed on the main user-interface thread, so the *updateGui* function can only be executed from such thread. The result should also be returned on the GUI thread (if the caller displays it in the user-interface immediately).
- Finally, the **remote** construct involves communication over the network and so the application would fail if it was executed in a sandbox that does not allow network communication (i.e. in a restricted mode on a phone).

When using types just to capture the data that functions take as arguments or return as the result, the type of the *dictionaryLookup* function is *string* $\rightarrow$ *string* (taking a word and returning its translation) and the type of *updateGui* is *unit* $\rightarrow$ *string* (taking no input, because the value is obtained via a resource, and returning a new text to display).

2.1.1 Problem statement

Using types to capture *how* expressions evaluate, we would like to express and statically check additional properties of computations. Thus the prob-

lem that we attempt to tackle is: *How can we use types to rule out runtime errors, such as the ones discussed in the previous section?*

To achieve that goal, we imagine that the type of functions could be annotated with additional information that specify in what context can a function be evaluated and what effects does a function have on the environment. The two functions from the previous section might be assigned the following types:

$$\begin{array}{ll} \mathit{dictionaryLookup} & : \mathit{Env}^{\{\mathit{server}\}}\ \mathbf{string} \rightarrow \mathbf{string} \\ \mathit{updateGui} & : \mathit{Env}^{\{\mathit{client}\}}\ \mathit{Cap}^{\{\mathit{network}\}}\ \mathbf{unit} \rightarrow \mathit{At}^{\mathit{gui}}\ \mathbf{string} \end{array}$$

In this example, the type constructor $\mathit{Prop}^{M}\ \tau$ is used to annotate a function type with additional information. When added to the input of a function, it specifies the context in which the function can run (for example, $\mathit{Env}^{\{\mathit{server}\}}$ means that the function can only be executed on the server). When added to the result of a function, it specifies the effect that the evaluation has (for example, $\mathit{At}^{\mathit{gui}}$ means that the result is returned on the main user-interface thread).

The tracking of effects using a type attached to the result of a function (using monads) is well known in the academic literature. We give more details about effects, monads and other related work in Section 2.2. The tracking of context-dependence is a new concept that has been developed by the author and is further discussed in Section 2.3.

There is a number of problems that remain unsolved. For example, different properties that may be attached to the type of computation propagate in different ways. It is also interesting to consider those from a logical perspective (via the Curry-Howard correspondence). The above example also used multiple different types (of form $\mathit{Prop}^{M}\ \tau$), but that assumes that the properties are composable, which is not, in general, the case for monads. Finally, effect (and coeffect) systems track just an approximation of the actual effects. Making the annotations more precise,. possibly using dependent typing, is another interesting future work.

2.2 Literature review

This section presents some of the existing work that is related to the aims described in the previous section. It starts by introducing effect systems and monads. Effect systems track one kind of properties of computations and monads provide, to some extent, unified framework for tracking effects (Section 2.2.1).

2.2.1 Effects and monads

Effect systems. Introduced by Gifford and Lucassen [12], effect systems have been designed to track effectful operations performed by a computation. Examples include tracking of operations on memory locations, communication in message-passing systems [16] and atomicity in concurrent applications [11]. Effect systems are usually described as typing judgements of the form $\Gamma \vdash e : \tau ! \sigma$, associating effects σ with the result. Effect systems are added to a language that already supports effectful operations, which is called *descriptive* approach by Filinski [9].

Monads. Purely functional languages use monads, introduced by Moggi and Wadler [27, 45], for a similar purpose. Monads provide an abstraction that allows embedding of imperative constructs in a pure setting. A definition of monad represents impure computation as some (pure) data type and defines composition of such computations. A computation producing a value of type τ in a monad M has a type $M\tau$. Filinski calls the approach where effects need to be implemented *prescriptive*. Note that the monadic type, in this case, does not specify what particular effects have been performed, which is usually the case with effect systems. The type $M\tau$ just specifies that the computation may perform any effect represented by the monad M.

Marriage of effects and monads. Wadler and Thiemann [47] showed that an effect system can be transposed to a corresponding monad system. This means that the two typing schemes are equivalent. The monadic type M needs to be parameterized with the particular effects σ, so a judgement $\Gamma \vdash e : \tau ! \sigma$ corresponds to a typing judgement $\Gamma \vdash e : M^{\sigma}\tau$.

2.2.2 Types for tracking contexts

Aside from the tracking of computational effects, we may also want to track in what context (or environment) may a computation execute. A context may be a specific node in a distributed application (i.e. server, phone or browser) or a security level associated with the caller (high or low).

Conversely to effect systems, typing judgements of type systems that track the dependence on a context are usually of the form $\Gamma @ \mathcal{C} \vdash e : \tau$, associating the context-dependent properties of e, specified by $\mathcal{C}$, with the typing assumptions or input. Unlike for effects, there is no common terminology for such systems, so we introduce the term *coeffect* system (Section 2.3).

Coeffect systems. There is a handful of type systems that could be classified as coeffect systems using the definition from the previous section. Many

of such systems track properties of a computation and it is an intriguing problem whether we can devise a common framework that would be powerful enough to capture some of them.

In the calculus of capabilities [7], the context carries *capabilities* that allow access to memory regions. The context is modified by memory allocation and deallocation. In a system for safe locking [10], the context carries a set of acquired locks (enabling access to a reference) and *synchronize* construct modifies the context.

Type systems that guarantee secure information flow [38, 44] can also be viewed as coeffect systems. The context specifies the secrecy of the information (high or low). Certain operations are only allowed when the context represents specific secrecy.

Our motivating example included distributed programming features akin to ML5 [29] or QWeSST [39]. Both of these languages mark code with a single environment where it is executed. The Links language [6] supports different execution environments (client, server, database) and uses a simple effect system to track whether database was accessed or not. Although this is written in the effect-style, the same property could be also tracked in the coeffect-style, which suggests that the two approaches may be dual.

Finally, implicit parameters by Lewis et al. [19] is an example that does not track properties of a computation, but instead tracks dynamically scoped parameters as part of the context.

2.2.3 Towards unified theory

Composing effect and coeffect systems. As mentioned earlier, effects can be tracked using monads. However, composing monads does not, in general, give a monad, so composing different monads that track different effects is an open problem. Solutions exist for certain kinds of monads [15, 20], but not for the fully general case.

One possible alternative is to use a weaker form of abstraction such as *applicative functors* [25]. As far as we are aware, the relation between effect systems and applicative functors has not been explored yet. Another approach to composing (or layering) of monads has been developed by Filinski [8] and can be efficiently implemented in a language that supports continuations.

Swamy et al. [40] present a lightweight approach that allows programming with multiple different monads related by morphisms that are automatically inserted. This approach can be used for working with monads that represent effects. A monad allowing read access to a memory location could be automatically lifted to a monad allowing read and write access (a computation that only reads memory can surely be executed in a monad that allows both reads and writes). When using the *prescriptive* approach, morphisms have to be implemented for every pair of monads. However, when

using the *descriptive* approach, morphisms become just identity functions.

So far, the work discussed in this section considers effects and monads. We demonstrate [35] that dependence on a context (or *coeffects*) can be tracked using *comonads*, a categorical dual of monads. A type system that tracks both effects and coeffects may need to compose computations based on monads and comonads, which is also an interesting topic for future work.

Parameterized monads and Hoare types As mentioned earlier, to track effects using a monadic type system, the monad needs to be parameterized with the performed effects σ, so we use types of form $M^{\sigma}\tau$. The tag can be formed by various algebraic structures.

Practical systems [1, 18] often use sets, typically with union to combine effects of sub-expressions. A simple alternative is to use a fixed set of values (i.e. to represent different threads [33]).

Parameterised monads of Atkey [1] use tags that comprise pre- and post-conditions of a computation. This presents a more general reasoning framework that could be used to reason about any kinds of effects. This may be especially useful in a dependently-typed functional language as the specifications could capture the effects precisely (possibly expressed as a predicate) as opposed to overapproximating the effects. As demonstrated by McBride [23], parameterized monads are also related to specifications in Hoare type theory [30], which is a general (dependently-typed) framework for static tracking of side-effects.

2.2.4 Modal and linear logics

Types in programming languages are related to proofs via the Curry-Howard correspondence. In particular, *linear logic* [14] gives rise to linear types that can be used to guarantee that resources are used exactly once [46]. Resource usage is an important property of a computation, so it is desirable to allow embedding of linear logic into a general framework for tracking the properties of a computation.

Type systems based on *modal logics* have been used in distributed programming [28, 29, 31] and their use has been also suggested for multi-stage computations [37]. In general, modal logics allow reasoning about possible worlds – in programming language terms, we can view possible worlds as different *contexts* of a computation.

The categorical model of intuitionistic modal logic developed by Bierman and De Paiva [3] is based on comonads. As mentioned earlier, a type system that we developed for tracking of coeffects is also based on comonadic calculus, which suggests that modal logics may be very useful for reasoning about *coeffects*.

$$\text{(let)}\frac{\Gamma \vdash e_1 : \tau_1, \sigma_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2, \sigma_2}{\Gamma \vdash \textbf{let}\, x = e_1\, \textbf{in}\, e_2 : \tau, \sigma_1 \cup \sigma_2} \qquad \text{(fun)}\frac{(\Gamma, x : \tau_1) \vdash e : \tau_2, \sigma}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\sigma} \tau_2, \emptyset}$$

$$\text{(colet)}\frac{\Gamma@r_1 \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1@r_2 \vdash e_2 : \tau_2}{\Gamma@r_1 \cap r_2 \vdash \textbf{let}\, x = e_1\, \textbf{in}\, e_2 : \tau} \qquad \text{(cofun)}\frac{(\Gamma, x : \tau_1)@r \vdash e : \tau_2}{\Gamma@\mathcal{R} \vdash \lambda x.e : \tau_1 \xrightarrow{r} \tau_2}$$

Figure 2.2: Example rules of effect and coeffect system side-by-side

2.3 Coeffects

This section briefly presents the work on coeffect type systems that was done by the author during the first year. The motivation for this work has been already outlined in Section 2.2.2. There is a range of various type systems that track how computations depend on the context. Typing judgements of such type systems are usually written in a form $\Gamma@\mathcal{C} \vdash e : \tau$, meaning that an expression e can be assigned a type τ when provided with free variables Γ and a context $\mathcal{C}$.

In this report, we briefly describe an example of a language that uses coeffect typing, relate it to effect typing, and then present core typing rules of our general type system for tracking context-dependence using comonads. More details can be found in a draft paper on this topic [35].

2.3.1 Effect and coeffect systems

When using effect system to track memory accesses, the effects are preformed by some primitive operations. To track effects precisely, reference cells are annotated with a tag specifying the memory region where the effect is performed. For example, a reference cell r of type $\textbf{ref}_\rho\tau$ stores a value of type τ in a memory region ρ. An expression $r := v$ that assigns a value v of type τ to the reference cell has an effect $\{w(\rho)\}$ specifying that the operation writes to the location ρ.

Effect system ensures that the information is correctly propagated through the program. For example, the rule *(let)* in Figure 2.2 shows that effects performed by two sub-expressions are combined using union of sets. A rule for lambda abstraction *(fun)* shows another interesting aspect of effect systems – the effects of function body are associated with the function type, so a function $\tau_1 \xrightarrow{\sigma} \tau_2$ represents a function that will perform effects σ when executed.

Coeffect system When using coeffect system to track possible execution environments in a distributed programming language, the context specifies the environments where an expression can evaluate. For example, an expression **access** *gui* (representing access to the user-interface) may be typeable only in context $\{\mathit{browser}, \mathit{phone}\}$, but not in the *server* environment.

When composing expressions that can be executed in two different execution environments *(colet)*, the contexts (representing sets of environments) are combined using intersection, because the resulting expression can only be evaluated in the environments where all sub-expressions can be evaluated. Similarly to effect systems, the *(cofun)* rule also associates the required context with the type of function, so for example, a function that reads user interface would have a type $\mathit{unit} \xrightarrow{r} \mathit{string}$ where $r = \{\mathit{browser}, \mathit{phone}\}$.

Although intuitively quite different, the systems for tracking effects and coeffects have striking similarities and appear dual. There is one notable difference. The *(cofun)* rule from Figure 2.2 is just a special case and the *comonadic type system* uses a more general and more expressible variant.

2.3.2 Comonadic type system

When modelling a language using categorical semantics, an expression e of type τ in a context $x_1 : \tau_1, \ldots, x_n : \tau_n$ can be modelled as a function $\overline{\tau} \to \tau$ where $\overline{\tau} = \tau_1 \times \ldots \times \tau_n$.

As shown by Moggi [27], many types of effects can be modelled using a monadic structure over the type of the result. This means that the semantics uses functions of type $\overline{\tau} \to M\tau$. Our comonadic type system is based on a categorical semantics of context-dependent computations of Uustalu and Vene [43]. Contrary to effects, a model of a context-dependent computation has an additional structure on the input parameter. The structure specifies the additional context that is needed to evaluate the expression. Thus a comonadic semantics uses functions of type $C\overline{\tau} \to \tau$ where C is a comonad.

To track the specific information about context at the type-level, we annotate the comonad with a tag that describes the context (just like monads are annotated with a tag that describes effects). The semantics then operates on functions $C^r\tau_1 \to \tau_2$. To make the framework more general, we use tags $r \in R$ that form a monoid $(R, 1, \otimes)$. This way, the tags can be formed by a set with union, or an intersection, but also other simple structures.

Typing rules. In the categorical semantics, the comonadic structure provides mechanism for composing context-dependent computations. We do not use the structure to give model to our language – instead, we use it to derive the judgements of a type system. For example, a composing a function $C^r\tau_1 \to \tau_2$ with a function $C^s\tau_2 \to \tau_3$ gives a function $C^{r\otimes s}\tau_1 \to \tau_3$,

$$\text{(cobind)}\ \frac{C^r\Delta \vdash e_1 : \tau_1 \quad C^s(\Delta, a:\tau_1) \vdash e_2 : \tau_2}{C^{r\otimes s}\Delta \vdash \textbf{let}\ a \Leftarrow e_1\ \textbf{in}\ e_2 : \tau_2} \qquad \text{(coapp)}\ \frac{C^r\Delta \vdash e_1 : C^t\tau_1 \to \tau_2 \quad C^s\Delta \vdash e_2 : \tau_1}{C^{r\otimes s\otimes t}\Delta \vdash e_1\ e_2 : \tau_2}$$

$$\text{(counit)}\ \frac{a:\tau \in \Delta}{C^1\Delta \vdash a : \tau} \qquad \text{(cofun)}\ \frac{C^{r\otimes s}(\Delta, a:\tau_1) \vdash e : \tau_2}{C^r\Delta \vdash \lambda^\circ a.e : C^s\tau_1 \to \tau_2}$$

Figure 2.3: Comonadic typing rules.

which corresponds to the **let** construct. The key rules of our comonadic type system are shown in Figure 2.3.

The *(cobind)* rule is similar to the rule for **let** construct that we discussed when introducing coeffects. The only difference is that the contexts r and s are combined using the $\otimes$ operation of a monoid, instead of specific set operation. The *(counit)* specifies that a variable access is performed in a *unit context*. This is a context with no restrictions or requirements (i.e. a set of all possible environments in a distributed programming language).

The *(coapp)* rule defines application of a comonadic (context-dependent) function $C^t\tau_1 \to \tau_2$. In order to apply the function, the context must be a combination of contexts required to evaluate the two expressions e_1 and e_2 and also the context in which the function can be executed.

Finally, the *(cofun)* rule in Figure 2.3 differs from an earlier *(cofun)* rule for distributed languages. In general, the function can be created in a context r and can take a parameter that carries context s (this is the context where the function is called). The body of the function is evaluated in a context $r \otimes s$ that is obtained as a combination of the two contexts.

Merging of contexts. The merging of contexts in the *(cofun)* rule is an interesting feature that does not directly correspond to any aspect of monadic effect systems. This suggests that the comonadic coeffect system is not entirely dual to monadic effect systems. For distributed programming language, the restricted *(cofun)* rule shown earlier is more appropriate, but there are other applications. For example, the generalized version is suitable model for implicit (dynamically scoped) parameters [19].

Formally, the merging of contexts is specified by an additional operation of a *monoidal comonad*. In addition to *cobind* (representing composition) and *counit* (representing a pure computation), a monoidal comonad also defines an operation that we call *combine*:

$$\mathit{combine} : C^r\tau_1 \times C^s\tau_2 \to C^{r\otimes s}(\tau_1 \times \tau_2)$$

The operation takes two values that carry different contexts and produces a single value, containing a tuple, associated with just a single context. In our categorical semantics, the operation is used to combine outer context (where a function is defined) with an inner context (carried by the parameter).

The tags in the above type signature allow both of the contexts to be arbitrary, which gives the general *(cofun)* rule from Figure 2.3. By specializing the tag of one of the two contexts to 1 (specifying *unit context*) we get two useful variants of the system – one where functions can not capture context in which they are defined (dual to monadic typing) and one that only allows declaration of pure functions.

We find it intriguing that formally dual concepts such as monad and comonad give rise to two type systems that are not dual in practice. The comonadic type system requires the use of *monoidal comonad*, which adds flexibility in how contexts are composed. This flexibility makes the system quite useful in practice – we do not have space to give more examples, but an interested reader is referred to our draft paper [35] for more details.

2.4 Joinads

Another work that was completed by the author during the first year introduces *joinads* [34]. The original motivation for joinads is quite different than the motivation of this report, but Section 3.3.3 shows that joinads may be relevant for the tracking of effects.

2.4.1 Introducing joinads

Monads can be used for sequencing of effectful computations and languages like Haskell and F# introduce notations (**do** notation [36] or *computation expressions* [41]) for writing sequential code with effects.

However, many concrete types that implement the monad structure also provide additional operations for composing computations in different ways. For example, the *Par* monad [22] adds operations that can be used to implement parallel composition of type $Par\ a \rightarrow Par\ b \rightarrow Par\ (a, b)$. Similar operation can be defined for other monads for concurrent or parallel programming, such as Communicating Haskell Processes [5]. The *Par* monad can be also extended [32] to support non-deterministic choice, which makes it possible to implement *speculative parallelism* pattern. The choice operator has a type $Par\ a \rightarrow Par\ a \rightarrow Par\ a$.

Many monads, even outside of the concurrent and parallel programming field, implement operations like the ones above. For example, monadic parsers [13] usually provide choice and parallel composition can be defined as the intersection of languages recognized by the parsers.

When monadic libraries provide additional ways of composing computations, they do so in an ad-hoc fashion, because there is no standard set of additional operations. Each monadic library adds slightly different set of operations. We introduce *joinad*, which is a monad that implements three additional operations (choice, parallel composition and aliasing). We also extended Haskell with a **docase** notation that bears similarity to **case** and can be used for programming with computations that implement the joinad structure.

2.4.2 Programming with joinads

The paper [34] provides numerous example computations that can implement the joinad pattern. In this report, we demonstrate joinads and the **docase** notation using a monad for parallel programming. First, consider the following function (written in Haskell) that tests whether a predicate holds for all leaves of a tree:

$$
\begin{array}{l}
\mathit{all} :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Bool} \\
\\
\mathit{all}\ p\ (\mathit{Leaf}\ a) \quad\quad\quad\quad = p\ v \\
\mathit{all}\ p\ (\mathit{Node}\ \mathit{left}\ \mathit{right}) \;\; = \mathit{all}\ p\ \mathit{left} \wedge \mathit{all}\ p\ \mathit{right}
\end{array}
$$

The execution of the two recursive calls in the *Node* case 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*.

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

$$
\begin{array}{l}
\mathit{all} :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Par}\ \mathit{Bool} \\
\\
\mathit{all}\ p\ (\mathit{Leaf}\ a) \quad\quad\quad\quad = \mathit{return}\ \$\ p\ v \\
\mathit{all}\ p\ (\mathit{Node}\ \mathit{left}\ \mathit{right}) \;\; = \\
\quad \textbf{docase}\ \mathit{all}\ p\ \mathit{left}, \mathit{all}\ p\ \mathit{right}\ \textbf{of} \\
\quad\quad \mathit{False}, ? \quad\quad \to\ \mathit{return}\ \mathit{False} \\
\quad\quad ?, \mathit{False} \quad\quad \to\ \mathit{return}\ \mathit{False} \\
\quad\quad \mathit{allL}, \mathit{allR} \quad \to\ \mathit{return}\ \$\ \mathit{allL} \wedge \mathit{allR}
\end{array}
$$

The function builds a computation annotated with hints that specify how to evaluate it in parallel using the *Par* monad [22] extended with the support for non-deterministic choice operator [32].

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 immediately enabled. After the right subtree is processed, one of the last two alternatives can be selected. The choice added to the *Par* monad is non-deterministic, so the programmer needs to provide clauses that produce the same result in case of race.

2.4.3 Joinad operations

As already mentioned, a joinad is a monad with three additional operations that represent *parallel composition*, *choice* and *aliasing*. The types of joinad operations are shown below:

$$
\begin{array}{ll}
\mathit{unit} & :: \ \tau \to M\tau \\
\mathit{bind} & :: \ (\tau_1 \to M\tau_2) \to M\tau_1 \to M\tau_2 \\
 & \\
\mathit{mzip} & :: \ M\tau_1 \to M\tau_2 \to M(\tau_1 \times \tau_2) \\
\mathit{morelse} & :: \ M\tau \to M\tau \to M\tau \\
\mathit{malias} & :: \ M\tau \to M(M\tau)
\end{array}
$$

When desugaring the previous example, the selection between alternative clauses is done using the *morelse* operator. Note that the result of each input computation is used in two independent alternatives. Evaluating the recursive computation repeatedly would defeat the purpose of **docase**, so the translation uses the *malias* operator to avoid this. For the *Par* monad, the *malias* operation starts the given computation in background and returns (inside a monad) a new monadic computation that blocks until the background computation completes (without consuming any CPU resources). Finally, the third alternative combines two computations, which is achieved using the *mzip* operator[2].

Effect interpretation. The use of joinads presented above is to implement non-standard notion of computation, such as parallel computations or parsers. This corresponds to the *prescriptive* style as described by Filinski [9]. As shown by the marriage between effects and monads [47], structures that implement computation can be also used to track properties of computations that already exhibit the effects (*descriptive* style).

The additional operations supported by joinads could be used to track effects in a language with additional expressive power. The *mzip* corresponds to running two effectful operations in parallel (a tagged version may restrict the arguments to guarantee that the effects do not operate on a

[2]For detailed description of the translation see [34]

shared state). The *morelse* corresponds to a choice between two alternative computations (for example, branches of the **if** construct). The meaning of the *malias* operator is less clear to us, but it might represent evaluating a part of effects (for example, in a language that supports partial evaluation).

Chapter 3

Thesis proposal

As discussed in Section 2.1, the aim of the thesis is to develop a comprehensive framework for statically checking various properties of computations using types. This is becoming an important topic as computations move to a distributed setting (running in the cloud) and applications needs to run on different kinds of devices (browser, phone, ...) or with different security permissions.

The pioneering work on effect systems and encoding of effects using monads (discussed in Section 2.2) provide a good starting point. However, we believe that there is a lot of work to be done in order to make tracking of effects (and similar properties such as coeffects) a prevalent feature of main-stream (statically typed) programming languages.

In this chapter, we give a brief overview of the open problems (Section 3.1). Then we outline a timeplan for the rest of the PhD (Section 3.2) and discuss some of the planned projects in greater detail (Section 3.3).

3.1 Towards practical computation types

Effect systems can be used to track properties of computations that propagate in the forwards direction. A computation that invokes another computation will inherit the effects of the invoked computation. In the work completed during the first year (Section 2.3), we introduced coeffects to track properties that propagate backwards (such as dependence on a context). To track properties of computations, we need to be able to combine these two approaches. Moreover, it is an open question whether effects and coeffects suffice to capture all kinds of properties or whether there is some other way in which properties propagate (i.e. based on other abstract computation types such as *applicative functors* [24]).

Effect and coeffect systems generally over-approximate the effects that a computation may perform. Traditional effect systems [12, 47] used sets of effects together with union to combine effects of sub-expressions. However,

in case of **if**, we might want to capture the fact that only one of the branches will be executed. Using more complex structure would make the approximation more precise at the cost of making the analysis more difficult. Taking the extreme approach, we might even want to track the *exact* effects that a computation might perform. This would lead to a dependently-typed system related to Hoare Type Theory [30]. For practical purposes, we believe that over-approximation of effects is the best approach, but we intend to explore other directions as well.

One of the aims of this thesis is to design a practically useful type system that might be non-intrusively integrated in a programming language such as F#. We intend to support defining of effects in a library – in particular, the developer should be able to define a simple algebraic structure used to track the effects (i.e. a monoid) and algebraic equations that can be used by type inference to derive effects and coeffect annotations. Adding type inference and polymorphism is important future work for our coeffect system described earlier. Ideally, we would like to design a language mechanism that allows us to define various effect and coeffect annotations, but is also capable of expression F# units of measure, preserving the non-intrusive style that is used in the current design.

3.2 Timeplan

During the next 2 years of my PhD, I intend to focus on three major projects. The first project is the integration of systems for tracking coeffects and effects of computations and extension that allows non-intrusive type-inference and polymorphism. The second project is to implement the system that allows developers to specify tracking of various kinds of effects and coeffects and perform evaluation using a wide range real-world application. Finally, the last project (to some extent independent) is to design the tracking of precise effects and coeffects in a dependently-typed language.

The major milestones that need to be completed in order to finish the PhD thesis are the following:

- **January 2012 – April 2012**: Study the relation between comonadic type system for tracking of coeffects (described earlier) and non-classical logics. The system has sub-structural rules, which suggests that it may be used to track the use of resources (based on linear logics). Moreover, type checking in different contexts is also closely related to possible worlds from modal logics.

- **April 2012 – September 2012**: Work on effect (and coeffect) systems that give more precise approximations of the performed effects or required context. Doing the work in a dependently-typed language

(such as F* [4]) would allow capturing precise specification of the effects. To our knowledge, this is an interesting and unexplored area.

- **September 2012 – February 2013**: After receiving feedback on the initial draft of the paper that introduces coeffects [35], I would like to expand this work in two ways. One is to make the system more realistic (by adding type inference and polymorphism) and the other is to develop a system that allows combining effects and coeffects in a single program. This may also give a compositional way of writing denotational or operational semantics of such programs.

- **February 2013 – July 2013**: As the last part of my research, I intend to implement a type system for tracking user-specified effects and coeffects as an extension of an ML-style functional language, such as F#. The resulting design should be evaluated by developing (or modifying an existing) system that could benefit from tracking of computation types. The application may use features outlined in Section 2.1, such as distributed programming, security sandbox and complex threading.

- **August 2013**: I intend to start writing up my thesis in August 2013 in order to submit it at the end of the third year of my PhD.

Aside from the key milestones outlined in the timeplan, I also intend to explore various additional side-problems that arise from the work. For example, the *malias* operation may be relevant to defining a *call-by-need* or *call-by-future* version of lambda calculus translation [45]. Another side-problem is whether the monad-based version of *joinads* can be used to encode join calculus (and how to combine such language with type-checking of distributed programming languages using coeffects).

3.3 Proposed thesis outline

This section discusses a possible thesis outline. It expands on some of the ideas for future work proposed in Section 3.1 and planned in Section 3.2. The following listing gives more details on individual chapters. It lists four chapters that present main novel research contributions, although the final thesis will likely contain only the three most interesting ones:

1. **Introduction** – describes the motivation and introduces the problem
2. **Background** – expands the discussion presented here, in Section 2.2
3. **Effect and coeffect types** – the chapter is based on a submitted work on coeffects [35] and discusses extensions, such as combining of multiple properties (Section 3.3.1) and type inference

4. **Coeffect logic** – discusses logical interpretation of coeffect type system (Section 3.3.2) and relation to modal and linear logics

5. **Precise effects and coeffects** – describes approaches to tracking more precise information about effects, using a more detailed algebraic structures or using propositions in a dependently-typed programming language (Section 3.3.3)

6. **Implementation and evaluation** – describes an implementation of a language extension for tracking of user-defined effects and coeffects and evaluates the extension using a non-trivial system.

7. **Conclusions**

3.3.1 Combining effects and coeffects

A well-known limitation of monadic types is that they do not (easily) compose. When encoding effects using monads, we may use two different monads to represent two different kinds of effects. A computation that performs both effects should have a type in some combined monad. For example, say we have monads $M_1\tau$ and $M_2\tau$ with the following operations:

$$\begin{array}{ll} \mathit{bind}_1 & : (\tau_1 \to M_1\ \tau_2) \to M_1\ \tau_1 \to M_1\ \tau_2 \\ \mathit{bind}_2 & : (\tau_1 \to M_2\ \tau_2) \to M_2\ \tau_1 \to M_2\ \tau_2 \\ \mathit{unit}_1 & : \tau \to M_1\ \tau \\ \mathit{unit}_2 & : \tau \to M_2\ \tau \end{array}$$

To represent computations that perform both effects in M_1 and M_2, we need to construct a composed type $M_1(M_2\ \tau)$. In general, this composed type is *not* a monad, because it is not possible to define a *bind* operation for this type just using the operations above. As discussed for example in [15], we need an additional operation such as:

$$\mathit{join}\ : M_1(M_2(M_1(M_2\ \tau))) \to M_1(M_2\ \tau)$$

Using the operations of M_1 and M_2 together with the additional *join* operation, it is now possible to define a combined monad $M_1(M_2\tau)$ as follows:

$$\begin{array}{ll} \mathit{return} & = \mathit{return}_1 \circ \mathit{return}_2 \\ \mathit{bind} & = \mathit{join} \circ \mathit{bind}_1\ (\mathit{return}_1 \circ (\mathit{bind}_2\ (\mathit{return}_2 \circ f))) \end{array}$$

When combining monadic effect systems and comonadic effect systems, we need to compose functions of type $C\tau_1 \to M\tau_2$. Finding the additional operations that are needed to compose multiple monads and comonads in a single computation is one of the problems that we intend to study.

Descriptive (co)effects. When using effect and coeffect types in the *descriptive* style, the types only annotate existing language that already implements accessing of context or performing of effects. In this mode, monadic (or comonadic) type can be viewed as just an identity monad (comonad), meaning that $M\tau = \tau$ and $C\tau = \tau$.

In this case, defining operations such as *join* is trivial. A model like this may be suitable for type systems that track properties of user-defined effects that can be already implemented in the underlying language. For ML-style languages such as F#, this includes threading, shared memory access as well as distributed programming.

Operational semantics. Numerous existing type systems can be described as a combination of coeffect and effect systems. For example, type system for safe locking [10] uses a context that carries a set of acquired locks. A reference cell protected by a lock l can be only accessed if the lock is in the context. This part of the system can be viewed as a *coeffect*. To take or release locks, the system provides an operation *synchronize* that modifies the context, which can be viewed as an *effect*. Additionally, accessing of reference cells can be viewed as another kind of *effect*.

The separation of the two aspects may be used to simplify the operational semantics of such system and make type soundness proof easier. In particular, the semantics [10] uses small-step reduction $\pi, \sigma e \mapsto \pi, \sigma, e$ where π represents the state of locks, σ stores values assigned to reference cells and e is the reduced expression. Our hypothesis is that the system could be more easily modelled using the following reductions:

$$
\begin{array}{ll}
e \mapsto e & (\textit{pure reduction}) \\
e \mapsto \pi, e & (\textit{lock-update reduction}) \\
e \mapsto \sigma, e & (\textit{memory-update reduction}) \\
\pi, e \mapsto e & (\textit{lock-protected reduction}) \\
\sigma, e \mapsto e & (\textit{memory-read reduction})
\end{array}
$$

The main simplification comes from the fact that each of the reductions has a pure expression e either as the input or as the result. The semantics might be constructed by composing these reductions (using a rule that corresponds to the *join* operation above). This would make it easier to prove safety of systems that compose multiple effectful or coeffectful aspects at the same time.

3.3.2 Logical interpretation

Our comonadic coeffect type system bears similarities to variants of lambda calculus that were designed for modal and linear logics [37, 3]. To study

the similarities, consider the following rule (adapted from [35]) that is used to type-check a context-dependent lambda function:

$$\text{(cofun)}\ \frac{C^{r\otimes s}(\Delta, a:\tau_1) \vdash e : \tau_2}{C^r\Delta \vdash \lambda^\circ a.e : C^s\tau_1 \to \tau_2}$$

The rule specifies that, in a context $C^r\Delta$ the term $\lambda^\circ a.e$ can be assigned a type $C^s\tau_1 \to \tau_2$. The body of the lambda function is type-checked in a context $r \otimes s$ that combines the context of the outer scope r, where the function is defined, and a context s that is passed to the function later as part of the argument. Using the Curry-Howard correspondence, the typing rule corresponds to the following logical judgement:

$$\frac{C^{r\otimes s}(\Delta, A) \vdash B}{C^r\Delta \vdash C^sA \to B}$$

In the premise, the left-hand side of a judgement consists of assumptions Δ and A and a context specified by $r \otimes s$. The context captures some additional property that specifies the particular information tracked by the type system. The *(cofun)* rule above was designed to give terms that correspond to a functional programming language. However, an interesting alternative direction is to separate the function abstraction (implication) and splitting of the context.

Instead of a single inference rule, we could use two inference rules that would allow writing the following derivation:

$$\frac{\dfrac{C^{r\otimes s}(\Delta, A) \vdash B}{C^r\Delta, C^sA \vdash B}}{C^r\Delta \vdash C^sA \to B}$$

Separating the implication and splitting of context may give an alternative formulation of the comonadic coeffect system that is more closely related to systems such as the λ^{S4} calculus [3]. The main difference is that in our original system, the context was associated with the entire free-variable context. Making splitting of context explicit means that a separate context may be associated with individual variables.

Multi-context formulation. When introducing comonadic coeffect system [35], we presented a version that combines two sub-languages. A plain lambda calculus can be used to write pure computations and a comonadic lambda calculus is used for writing context-dependent computations. The motivation for this design is practical – we might want to take a program in a pure language and gradually introduce coeffect types to track context-dependence as needed.

The typing judgement is written as $C^r\Delta; \Gamma \vdash e : \tau$ and it uses two distinct free varibale contexts. The context Γ contains pure variables and Γ contains

context-dependent variables. The additional context is associated with the free-variable context Δ using a comonadic type.

The formulation using two distinct variable contexts has been used in other logics. In particular, Bierman and de Paiva [3] present a multi-context formulation of an intuitionistic modal logic. One context carries only modal assumptions of the form $\Box\, A$. Benton uses similar approach to combine linear and non-linear logic in [2].

3.3.3 Precise effects

As discussed earlier, common (co)effect systems give just an approximation of the effects that will be actually performed by the program at runtime. For example, the typing rule for **if** might take the union of effects of the two branches, although they cannot be both executed at the same time. In the following, we write $\otimes$ for a union of sets forming a simple monoid $(\mathcal{F}, \cup, \emptyset)$:

$$\frac{\Gamma \vdash e_1 : \mathit{bool}@\sigma_1 \quad \Gamma \vdash e_2 : \tau@\sigma_2 \quad \Gamma \vdash e_3 : \tau@\sigma_3}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau@\sigma_1 \otimes \sigma_2 \otimes \sigma_3}$$

The tracking of effects could be more precise if we used a more complex algebraic structure than a monoid. For example, a semiring adds an additional operation $\oplus$ that could be used to represent a choice. A modified typing rule for **if** is:

$$\frac{\Gamma \vdash e_1 : \mathit{bool}@\sigma_1 \quad \Gamma \vdash e_2 : \tau@\sigma_2 \quad \Gamma \vdash e_3 : \tau@\sigma_3}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau@\sigma_1 \otimes (\sigma_2 \oplus \sigma_3)}$$

Note that a semiring specifies that $\otimes$ distributes over $\oplus$. For **if**, this means that $\sigma_1 \otimes (\sigma_2 \oplus \sigma_3) = (\sigma_1 \otimes \sigma_2) \oplus (\sigma_1 \otimes \sigma_3)$. This is the case for effects as $\oplus$ represents a choice and $\otimes$ represents sequencing of effects. Extending the framework to use semirings instead of simple monoids seems desirable.

Dependently-typed effects. In a dependently-typed language, such as F*, we could make one more step and track the effects *precisely* instead of calculating just an approximation. The standard rule for composing effects in a **let** expression is the following:

$$\frac{\Gamma \vdash e_1 : \tau_1@\sigma_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2@\sigma_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ : \tau_2@\sigma_1 \otimes \sigma_2}$$

Again, this is just an approximation, because the actual effects that will be performed when evaluating the expression e_2 depend on the value of x. In a dependently-typed language, the effect (which is a part of type) may depend on a value. If we write $\Pi x : \tau.\sigma$ for an effect that depends on value x of type τ, we could write a precise rule for **let** as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 @ \sigma_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 @ \sigma_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ : \tau_2 @ \sigma_1 \otimes (\Pi x : \tau_1 . \sigma_2)}$$

Interestingly, the algebraic structure that might be used to represent effects in this scenario seems very related to *joiands* from Section 2.4. A value dependent effect corresponds to the *bind* operation (that takes a function and returns an effect specification depending on a value). The operations $\otimes$ and $\oplus$ that were also defined for joinads (forming a near-semiring) can still be used to represent sequencing of effects and possibly a (non-deterministic) choice between effects, respectively.

3.3.4 Practical (co)effect systems

The aim of this thesis is to design a practically useful, extensible type system for tracking of properties of computations. The research described so far should provide a solid basis for developing such system, but there are several practical concerns.

In order to be useful, the system must not require developers to write type annotations for computation types explicitly. This can be achieved using an extended Hindley-Milner type inference algorithm. Similar extension has been already developed for lightweight monadic programming [40].

More importantly, the system must also infer and type-check the *tags* used to specify (co)effects. These may be sets of labels, more generally a monoid, or another more complex algebraic structure. Ideally, the developer of a library that provides some effect or coeffect should be able to specify the structure. This might be achieved using a mechanism based on F# type providers [26].

Units of Measure. An example of algebraic structure that has been successfully integrated in an ML-style functional language are units of measure in F# [17]. Numeric types in F# can carry an annotation that specifies their units. For example, the type *float*$\langle m/s \rangle$ represents a floating-point number in meters per second.

Using algebraic rules, the compiler can determine that the above type is the same as, for instance, *float*$\langle m^2 s / s^2 m \rangle$. Similarly, if we represented (co)effects using an idempotent monoid, the compiler should deduce that $C^{s \otimes s \otimes 1 \otimes r}\tau$ is the same type as $C^{s \otimes r}\tau$.

The type system also needs to provide polymorphism over (co)effect tags with a generalization that infers the most general type (if it exists). This can be, again, inspired by F# units of measure. For example, consider a function that takes a parameter a and returns $a * a * 1.0\langle m \rangle$. The type inferred by the compiler is *float*$\langle u \rangle \rightarrow$ *float*$\langle u^2 m \rangle$ where u is a type-variable representing any unit. A similar result would be desirable for other algebraic structures that might be defined to track computational properties.

Applications. There is a wide range of potential applications that could be used to assess the design. Classical effect systems [42] have been used to track operations on shared memory locations, which is still an important problem (i.e. when working with mutable arrays). A type system that would be also able to capture sub-structural rules could guarantee that such mutable state is accessed in a referentially transparent way.

The type system for safe locking [10] and safe asynchronous workflows in F# [33] are two examples of applications from the very important domain of concurrent programming, however we believe that there is a potential for numerous other applications.

Another possible application area is security – the system might be used to statically determine what capabilities does an application require when executed in a sandbox (i.e. access to local computer resources). Coeffect typing can be also used to track the secrecy of information [44, 38], which would be especially valuable when combined with distributed programming (where data is transferred over an unsecured network connection).

Chapter 4

Conclusions

The PhD thesis proposed in this report is centred around the idea of tracking of computation properties using types in an ML-style functional programming language. We argued that this is an important topic – modern applications run in various environments (different nodes of a distributed system or protected in a sandbox) and perform various effects (including memory access, communication and other concurrency-related actions). The overall aim of the thesis is to develop a unified framework that can be used to easily extend type-system to allow tracking of such properties (preferably via a user-defined library).

We describe work done during the first-year. Most notably, we briefly introduced a comonadic type system for tracking of *coeffects*, the dual notion to effects, that specify how computations depend on the context.

We also outlined some of the work that remains to be done. Two of the projects are closely related to coeffects. We intend to study the type system from a logical perspective and relate it to modal and linear logic. Next, we plan to develop ways for combining multiple effect and coeffect systems in a single language, allowing us to track multiple properties of a single program.

Two other projects are more implementation-focused. One aims to make the tracking of effects more precise by refining the algebraic structure that represents them. Using a dependently-typed language, we might be able to express and track the *exact* (co)effects of a computation. Finally, the last task is to integrate the research in a real-world functional language such as F# and evaluate it using a realistic application.

Bibliography

[1] R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19:335–376, July 2009.

[2] N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *Selected Papers from the 8th International Workshop on Computer Science Logic*, pages 121–135, 1995.

[3] G. Bierman and V. De Paiva. On an intuitionistic modal logic. *Studia Logica*, 65(3):383–416, 2000.

[4] J. Borgstrom, J. Chen, and N. Swamy. Verifying stateful programs with substructural state and hoare types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 15–26, 2011.

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

[6] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *In 5th International Symposium on Formal Methods for Components and Objects (FMCO*. Springer-Verlag, 2006.

[7] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, 1999.

[8] A. Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 175–188, 1999.

[9] A. Filinski. Towards a comprehensive theory of monadic effects. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 1–1, 2011.

[10] C. Flanagan and M. Abadi. Types for Safe Locking. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, 1999.

[11] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, 2003.

[12] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 28–38, 1986.

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

[14] Jean-Yves and Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

[15] M. P. Jones and L. Duponcheel. Composing monads. Technical report, Yale University, 1993.

[16] P. Jouvelot and D. K. Gifford. Communication Effects for Message-Based Concurrency. Technical report, Massachusetts Institute of Technology, 1989.

[17] A. Kennedy. Types for units-of-measure: theory and practice. In *Proceedings of the Third summer school conference on Central European functional programming school*, CEFP'09, pages 268–305, 2010.

[18] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 1–12, New York, NY, USA, 2008. ACM.

[19] J. R. Lewis, M. B. Shields, E. Meijert, and J. Launchbury. Implicit parameters: dynamic scoping with static types. In *POPL00*, 2000.

[20] C. Lüth and N. Ghani. Composing monads using coproducts. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 133–144, 2002.

[21] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: better strategies for parallel haskell. In *Haskell Symposium*, 2010.

[22] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, 2011.

[23] C. McBride. Kleisli arrows of outrageous fortune. Under consideration for publication in J. Funct. Program., 2011.

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

[25] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18, January 2008.

[26] Microsoft. Type Providers (F#). Available online[1], 2011.

[27] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.

[28] T. Murphy, VII, K. Crary, R. Harper, and F. Pfenning. A Symmetric Modal Lambda Calculus for Distributed Computing. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 286–295. IEEE Computer Society, 2004.

[29] T. Murphy, VII., K. Crary, and R. Harper. Type-safe distributed programming with ML5. In *Proceedings of the 3rd conference on Trustworthy global computing*, TGC'07, pages 108–123, 2008.

[30] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.

[31] S. Park. A modal language for the safety of mobile values. *Programming Languages and Systems*, pages 217–233, 2006.

[32] T. Petricek. Explicit speculative parallelism for haskell's par monad, 2011.

[33] T. Petricek. Safer asynchronous workflows for GUI programming. Available Online, 2011.

[34] T. Petricek, A. Mycroft, and D. Syme. Extending Monads with Pattern Matching. In *Proceedings of Haskell Symposium*, Haskell 2011.

[35] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: typing context-dependent computations using comonads. Submitted to ESOP 2012, 2012.

[36] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[37] F. Pfenning and H. Wong. On a Modal [lambda]-Calculus for S41. *Electronic Notes in Theoretical Computer Science*, 1:515–534, 1995.

[1] http://msdn.microsoft.com/en-us/library/hh156509(v=vs.110).aspx

[38] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 21(1), 2003.

[39] T. Sans and I. Cervesato. QWeSST for Type-Safe Web Programming. In *Third International Workshop on Logics, Agents, and Mobility*, LAM'10, Edinburgh, Scotland, UK, 2010.

[40] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight Monadic Programming for ML. In *Proceedings of the 2011 International Conference on Functional Programming*, ICFP '11, 2011.

[41] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 2.0*. Apress, Berkely, CA, USA, 1st edition, 2010.

[42] J. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162–173. IEEE, 1994.

[43] T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203:263–284, June 2008.

[44] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[45] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, 1990.

[46] P. Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[47] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.