A demo of Coco: a compiler for monadic coercions in ML

Nataliya Guts[†] Michael Hicks[†] Nikhil Swamy[*] Daan Leijen[*]
[†]University of Maryland, College Park [*]Microsoft Research, Redmond

Abstract

Combining monadic computations may induce a significant syntactic overhead. To allow monadic programming in direct style, we have developed Coco, a type-based tool that automatically rewrites ML code inserting necessary binds, unit, and morphisms between monads. This tool demonstration will show how to take advantage of Coco to facilitate using monadic libraries in practice, and will discuss possible future development of Coco to fit the actual needs of programmers.

1. The goal

Many functional programming computations may be expressed using monads [11]. Examples include probabilistic computations [7], parsers [6], functional reactivity [1], and cooperative threads [9]. ML provides some features that can be expressed monadically, like state and I/O, at no cost to the programmer, in a direct style. However, monadic computations such as those listed above are not directly supported. Instead, their implementation and composition must be tediously orchestrated by the programmer.

For example, suppose we are interested in programming *behaviors*, which are computations whose value varies with time, as in *functional reactive programs* [1, 2]. Behaviors can be implemented as a monad: expressions of type $Beh\ \alpha$ represent values of type α that change over time, and `bindp` and `unitp` are its monadic operations. As a primitive, function `seconds` has type $unit \rightarrow Beh\ int$, its result representing the current time in seconds since the epoch. We would like to be able to write programs using behaviors in direct style, for example:

```
let y = is_even (seconds()) in
if y then 1 else 2
```

The type of this entire expression should be $Beh\ \ int$: it is time-varying, oscillating between values 1 and 2 every second. However, in ML the above code is not type correct. For example, the function `is_even` expects an argument of type int, and it is applied to a computation of type $Beh\ \ int$. To make this example well-typed, we would need to insert explicit bind and unit operations, as in:

```
bindb (seconds ()) (fun s ->
   unitb (let y = is_even s in if y then 1 else 2))
```

which makes the program much harder to read and write.

There is some existing support for programming monads in a direct style. Haskell *do*-notation, or the *pa_monad* in OCaml provide special syntax for programming with monadic computations. In Haskell for example, we would need to bind the result of `seconds ()` in order to apply the `is_even` function:

```
do s <- seconds()
     let y = is_even s in
     return (if y then 1 else 2)
```

and similarly in the `pa_monad`:

```
perform
   s <-- seconds();
   let y = is_even s in
   return (if is_even s then 1 else 2)
```

Nonetheless, there is still some syntactic overhead in writing monadic expressions. Moreover, when multiple monads are involved where the programmer needs to explicitly lift one monad into another.

And despite the effort we needed to put into rewriting the example in a monadic style, there is actually little payoff! Since ML already has a built-in evaluation order, the original example can be naturally understood given its monadic type, and the translation to the monadic style can be done mechanically. This is how our tool, Coco, can automatically rewrite the example into the following code which has type $Beh\ \ int$ (modulo some simplifications mentioned in the next section):

```
bindb
  (bindb (seconds()) (fun s-> unitb (is_even s)))
  (fun y-> unitb (if y then 1 else 2))
```

Filinski [3, 4] gives monadic semantics to direct-style ML programs, relying on an implementation mechanism called *monadic reflection*. However, monadic reflection has not been integrated with ML polymorphic type inference, and requires monads to be implemented in a particular style.

2. Our approach: Type-directed rewriting

We adopt a different approach: program rewriting into monadic style based on an expressive type inference algorithm. We have developed Coco, a compiler for monadic coercions that converts ML programs into monadic style. Accordingly, given an ML program, Coco infers its type and inserts binds, units and morphisms where necessary, putting forward the monadic structure of the program. Our solution might be more adequate for several reasons. First, our approach is more expressive, in particular we support polymorphism. Second, we make no assumptions on how the monadic libraries are constructed, using them as typed black boxes. Finally, the programmer can actually access the rewritten program and its type. Coco may be particularly useful for retrofitting existing code into monadic style; for example if only a few dependencies' types have changed, Coco will propagate the changes through all the code.

The type-inference algorithm underlying Coco has been described in a separate paper [8]. Our demonstration will focus on the practical use of Coco and highlight some points of interest that came up in the process of its implementation. We will describe how Coco rewrites a small but non-trivial program that uses two distinct monads.

Figure 1 shows the architecture of Coco, its inputs and outputs. At present, Coco inputs a single file (*program.coco*) that includes the program to convert, and the type interface of the (monadic) val-

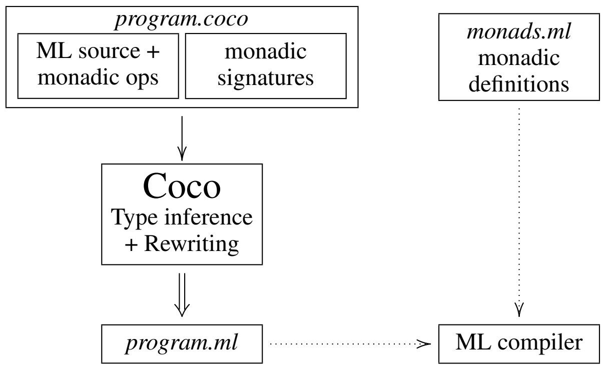

Figure 1: The Coco compiler

ues the program depends on, along with the monad declarations and available morphisms. The syntax of the source language and of the types is very close to that of OCaml. The monad declarations must contain a triple *(monad constructor, bind, unit)* for each monad, as well as possible morphisms between different monads. Coco outputs the rewritten well-typed source code, including necessary monadic operators and morphisms. Coco supports the option of outputting OCaml, which we will use for the demonstration. The generated code (*program.ml*) can run against an implementation of the monadic signatures (*monads.ml*).

Our type inference algorithm [8] introduces a fresh monad variable at applications and non-value let-bindings, and at values that are used as computations. The former are then sequenced using the monadic bind, and the latter are lifted using the monadic unit of the resulting monad. Monad variables are related by constraints which correspond to morphism application in the expressions. Solving these variables yields a well-typed program rewriting along with its type. Coco runs in linear time.

The procedure described in above is comparable to what the Haskell type-class mechanism does. However, on its own, this procedure is inadequate for many common programs that use our inference procedure. The problem is that the constraints we infer may admit many possible solutions, each yielding different semantics. As described in [8], Coco implements a custom constraint solving algorithm that exploits the monadic and morphism laws to resolve this ambiguity problem. Whenever a monad variable is the result of conversion of several distinct monads, and can be converted into another monad, our algorithm picks the least upper bound of its dependencies. This corresponds to applying conversions as late as possible, but is equivalent to applying the conversions in any possible order, thanks to the morphism laws. Using this algorithm, Coco can infer unambiguous types for all 12 programs in our benchmark suite. In contrast, a naïve implementation based on type classes would have rejected 19 types.

Coco supports two solving modes: in the *strict* mode, all top-level expressions must be annotated, and in the *permissive* mode, we admit the least upper bound of the dependencies as solution for monad variables which are not further constrained, which greatly simplifies the output types. Alternatively, we have implemented the "let should not be generalized" strategy for inner lets [10], which in practice simplified all of our examples.

The regular ML computations are encapsulated within a monad `Bot`, whose unit is simply the identity, and bind is the reverse apply function. In the implementation we simplify away most of the operators related to the `Bot` introduced by the algorithm.

To implement the OCaml generation feature, we had to encode higher-kinded resulting programs in OCaml. Indeed, a value let-binding yields a higher-kinded function parameterized with monads and monadic operators. We have encoded this using first-class modules in OCaml 3.12. Every monad is represented as a module with the standard monad signature. A let-bound value is encoded as a functor depending on some monad module variables. As a free benefit, we get to use first-class polymorphism for function arguments, which represent monadic operators.

3. Future directions

Coco is still at the development stage, and several directions for further implementation are possible. We will appreciate all feedback from the audience to get insights into what the community needs might be . In particular, for now we only support a subset of OCaml as input language. Depending on the demand among the programmers, we might to leave it at the stage of prototype, or carry on the implementation to support rewriting a full-fledged programming language. In the latter case, we consider several target languages: OCaml, or a language with native support of higher kinds. Lastly, we consider extending our technique for a dependently-typed language.

To sum up, this demonstration will aim to show that type-based rewriting is a convenient and programmer-friendly technique for making the use of monadic libraries painless. Our tool, Coco is an open-source prototype available for download [5], which we hope the audience will be willing to test.

Acknowledgements The authors would like to thank Gavin Bierman and Matt McCutcheon for their early contributions to this work. Hicks and Guts were both supported by NSF grant CNS-0905419.

References

[1] Greg Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, 2006.

[2] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[3] A. Filinski. Monads in action. In *POPL*, pages 483–494, 2010.

[4] Andrzej Filinski. Representing monads. In *POPL*, 1994.

[5] Nataliya Guts, Michael Hicks, Nikhil Swamy, and Daan Leijen. Coco. http://research.microsoft.com/en-us/projects/coco/.

[6] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *JFP*, 8(4), 1998.

[7] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

[8] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *ICFP*, 2011.

[9] J. Vouillon. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12. ACM, 2008.

[10] D. Vytiniotis, S. Peyton Jones, and T. Schrijvers. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 39–50. ACM, 2010.

[11] Philip Wadler. The essence of functional programming. In *POPL*, 1992.