Delimited Continuations and Monads: Exploring Abstractions from First Principles
Submitted for WPE-II examination April 5, 2021 by Harrison Goldstein.

Committee: Stephanie Weirich (chair), Benjamin Pierce (adviser), Rajeev Alur

This report is intended for a general computer science research audience (including early PhD students), and assumes basic familiarity with programming languages concepts (e.g., CIS 500 or equivalent).

Delimited Continuations and Monads*

Exploring Abstractions from First Principles

HARRISON GOLDSTEIN, University of Pennsylvania, USA

In 1990, two programming abstractions were introduced independently: *delimited continuations* and *monads*. These ideas quickly gained popularity, and over the course of 30 years they have permeated programming languages literature. Modern researchers often assume knowledge of these abstractions when writing papers, a practice which streamlines exposition but increases barriers to entry.

This report presents both delimited continuations and monads from first principles by following the papers that originally popularized the ideas. First I discuss *Abstracting Control* by Danvy and Filinski [2], and then I explore *Comprehending Monads* by Wadler [16]. Though the two abstractions initially seem to have little to do with one-another, delimited continuations and monads actually have a lot in common: they can implement many of the same design patterns, and their meta-theories are surprisingly compatible. I leverage these connections to develop intuition and provide context for future reading.

1 INTRODUCTION

Abstractions are the most powerful tools in a computer scientist's toolbox. They separate high-level ideas from low-level details, highlighting the novel and interesting parts of a program. Sometimes a particular abstraction becomes so ingrained in a community that members naturally think in terms of the abstraction and use it without explanation. Broadly, this is good—a powerful common language makes it easier to communicate ideas—but it does make it more difficult for outsiders and new researchers to understand a body of work. This report sets out to explain a couple of these abstractions for researchers familiar with, but not steeped in, the programming languages literature.

The notion of *delimited continuations* is one such abstraction that is ingrained in the programming languages community. These operators were first presented in *Abstracting Control* by Danvy and Filinski [2], and they are one of the many tools used to manipulate program continuations.

A *continuation* is a first-class representation of the *context* around a computation. For example, when evaluating the sub-expression "2" in the program

$$\text{let } i = 84 \text{ in let } j = 2 \text{ in } i/j,$$

we say the context is

$$\text{let } i = 84 \text{ in let } j = \_ \text{ in } i/j,$$

and the continuation, k, is

$$k = \lambda x.\, \text{let } i = 84 \text{ in let } j = x \text{ in } i/j.$$

The continuation captures the context as a function, which can be used as a first-class value. Danvy and Filinski present two operators for manipulating continuations: "shift k in e" and "reset e." The reset operation delimits a context, and shift packages that context as a continuation. For example, we can use shift and reset to extract the continuation from above:

$$\text{reset } (\text{let } i = 84 \text{ in let } j = (\text{shift } k \text{ in } k\ 2) \text{ in } i/j) \Rightarrow$$
$$\text{let } k = (\lambda x.\, \text{let } i = 84 \text{ in let } j = x \text{ in } i/j) \text{ in } k\ 2$$

*This report was compiled for part of the University of Pennsylvania's WPE-II Exam. The accompanying talk is available on the author's website.

Author's address: Harrison Goldstein, hgo@seas.upenn.edu, University of Pennsylvania, Philadelphia, PA, USA.

We replaced the expression 2 with shift k in k 2, which says "capture the context as k and then apply that to 2," and we wrapped the whole computation in reset to delimit the context. The result is the same as the original expression, but the continuation is made explicit. In that example, the shift body just applies k, but it can do anything. In particular, if k is not used at all,

$$\text{reset } (\text{let } i = 84 \text{ in let } j = (\text{shift } k \text{ in Nothing}) \text{ in } i/j)$$

the expression essentially throws an *exception*: the context is discarded and the result of the entire computation is just Nothing. This flexibility makes delimited continuations powerful but also notoriously difficult for newcomers to reason about.

Another abstraction that is deeply embedded in programming languages literature is the *monad*. Monads are notorious for being explained poorly, to the point where people jokingly quote "a monad is just a monoid in the category of endofunctors"—a sentence which is incomprehensible even to most community insiders [7, 9]. The internet is filled with monad tutorials that compare the mathematical structure to burritos (tasty, but unhelpful) and other "more accessible" concepts, but few provide useful explanations. To avoid falling into this trap of explaining monads badly, I follow here one of the earliest and clearest explanations of monads for programming, *Comprehending Monads* by Wadler [16].

Wadler presents monads using an analogy to list comprehensions like

$$[(x, y) \mid x \leftarrow [1, 2];\ y \leftarrow [3, 4]],$$

which evaluates to the Cartesian product of the lists [1, 2] and [3, 4]. Wadler observes that comprehension syntax can be used with more than just lists—in fact it can be used with any monad. One of the simplest monads is, called Maybe is defined as

$$\text{data Maybe } \alpha = \text{Nothing} \mid \text{Just } \alpha$$

in languages like Haskell. We can write a *Maybe comprehension* like,

$$[i/j \mid i \leftarrow [84]^{\text{Maybe}};\ j \leftarrow \text{if } n = 0 \text{ then Nothing else } [n]^{\text{Maybe}}]^{\text{Maybe}}$$

which uses comprehension syntax with the Maybe monad to abstract over *error handling*. In the expression, we first bind i to the value 84. Then we check if n is 0: if it is, the whole expression results in Nothing, otherwise we bind j to n. Finally, if the computation has not failed, we compute i/j. Just as list comprehensions provide a way to sequence lists, Maybe comprehensions provide a way to sequence computations that might fail. There are dozens of monads that programmers use in practice, each of which gives a different interpretation of comprehension syntax.

Both delimited continuations and monads are interesting on their own, but why present them together? It is not immediately clear that manipulating continuations has anything to do with interpreting comprehension syntax; one could be forgiven for assuming that these ideas are largely orthogonal. But we have already seen that delimited continuations and monads can both be used to simulate exceptions, and it turns out that there is significantly more overlap. Design patterns like state, nondeterminism, and more can be implemented using both delimited continuations and monads. Furthermore, continuations actually form a monad, and Wadler shows that this fact can be exploited to recover results from Danvy and Filinski.

In this report I make three contributions:

- I explain delimited continuations and monads by following two influential papers: *Abstracting Control* by Danvy and Filinski (§ 2) and *Comprehending Monads* by Wadler (§ 3).
- I develop intuition for programming with delimited continuations by implementing a number of the design patterns that Wadler presents as monads. This solidifies the close relationship between the two language features (§ 4).

- I present the *continuation monad* and show that, by performing a monad-agnostic transformation given by Wadler, we can recover the "extended continuation passing style" transformation presented by Danvy and Filinski (§ 5). This is a small extension of Wadler's original result.

I conclude with references to further reading that builds on these abstractions (§ 6).

Notation

My source papers differ slightly in notation and conventions. I have chosen a middle-ground and use one unified set of conventions to discuss both papers. Throughout this report, programs are written in a lambda calculus with syntax defined by:

$$\begin{aligned} e ::= {} & x \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid n \mid e_1 + e_2 \\ & \mid (e_1,\ e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \cdots \end{aligned}$$

The core calculus is made up of variables, x, lambda abstractions, $\lambda x.\, e$, and function application $e_1\ e_2$. Let-binding makes presentation clearer, so I assume we have that too, although formally I assume that

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

is implemented as

$$(\lambda x.\, e_2)\ e_1,$$

so I often ignore it in meta-theory. Danvy and Filinski use Booleans and natural numbers to illustrate delimited continuations; Wadler uses pairs and lightweight data structure definitions to illustrate monads—I also use these features as needed. Since we are almost exclusively concerned with dynamic semantics, I do not formalize a type system.

Built-in language operators are written in blue sans-serif and defined functions are written in black sans-serif.

2 *ABSTRACTING CONTROL*

Before explaining delimited continuations, I should explain some background on continuations and first-class continuation operators. Continuations first appeared as meta-theoretical tools [15], but before long they were introduced into concrete languages via the *continuation-passing style* or CPS translation. Here is a call-by-value CPS translation, $C[\![\cdot]\!]$, for our basic lambda calculus with Booleans and if-statements:

$$\begin{aligned} C[\![x]\!] &= \lambda\kappa.\, \kappa\ x \\ C[\![\lambda x.\, e]\!] &= \lambda\kappa.\, \kappa\ (\lambda x.\, C[\![e]\!]) \\ C[\![e_1\ e_2]\!] &= \lambda\kappa.\, C[\![e_1]\!]\ (\lambda f.\, C[\![e_2]\!]\ (\lambda x.\, f\ x\ \kappa)) \\ C[\![\mathsf{true}]\!] &= \lambda\kappa.\, \kappa\ \mathsf{true} \\ C[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] &= \lambda\kappa.\, C[\![e_1]\!]\ (\lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ C[\![e_2]\!]\ \kappa\ \mathsf{else}\ C[\![e_3]\!]\ \kappa) \end{aligned}$$

This translation extracts every evaluation context—essentially every individual sub-term of a program—as an explicit continuation function. The resulting terms do not "return" in the traditional sense: when these terms finish computing, they pass the resulting value to a continuation, κ, which is provided as an argument. A translated term can be run by applying it to the trivial continuation, and $C[\![e]\!]\ (\lambda x.\, x)$ should have the same semantics as e. In other words, this translation is semantics-preserving—it only changes the structure of the program.

The CPS translation has been used extensively in compilers and interpreters, where it gives fine-grained control over evaluation. This is because the translation makes evaluation order explicit. This particular CPS translation is clearly call-by-value, since in the rule for application,

$$C[\![e_1\ e_2]\!] = \lambda\kappa.\, C[\![e_1]\!]\ (\lambda f.\, C[\![e_2]\!]\ (\lambda x.\, f\ x\ \kappa))$$

the argument e_2 is evaluated before f is applied.

As CPS translations became more popular, researchers began to consider first-class abstractions for working with continuations. The most famous of these early abstractions, "call with current continuation," or call/cc [5], can be added as an operator in our source language and implemented as an extension to the CPS translation:

$$C[\![\text{call/cc } k \text{ in } e]\!] = (\lambda\kappa.\, C[\![e]\!]\ \kappa)[k \mapsto \lambda x.\, \lambda\kappa'.\, \kappa\ x],$$

where $e[x \mapsto v]$ means e with v substituted for free instances of x.

The call/cc operator is extremely powerful. The expression call/cc k in e captures the current program context as k and then runs e, allowing the programmer to arbitrarily pause and resume evaluation as they see fit. For example,

$$\text{call/cc } k \text{ in } 1$$

aborts the computation, returning the value 1, and

$$\text{call/cc } k \text{ in } (\text{print "foo"};\ k\ 10)$$

pauses the computation, prints "foo", and then resumes computation with the value 10. Unfortunately, many have argued that call/cc is *too* powerful. Consider the popular Yin-Yang Puzzle:

$$\begin{aligned}&\text{let yin } = (\lambda c.\, (\text{print "@"};\ c))\ (\text{call/cc } k \text{ in } k) \text{ in}\\ &\text{let yang } = (\lambda c.\, (\text{print "*"};\ c))\ (\text{call/cc } k \text{ in } k) \text{ in}\\ &\text{yin yang}\end{aligned}$$

What does this confusing mess of continuations do? Apparently it counts. This program prints increasing numbers in a unary representation,

$$\text{"@*@**@***@****@*****..."},$$

but it is extremely difficult to understand exactly why that is.

Felleisen [3] reigned in the power of call/cc with operators that he called prompt and control. These allowed programmers to delimit the effects of call/cc with *scopes*. Unfortunately, Felleisen's scopes were dynamic, and his approach did not admit a straightforward translation into a standard lambda calculus. This was the impetus for Danvy and Filinski's statically delimited continuations.

2.1 Delimited Continuations and ECPS

In *Abstracting Control*, Danvy and Filinski introduce the shift and reset operations mentioned in Section 1, which provide a more usable alternative to call/cc. We extend our lambda calculus:

$$e ::= \cdots \mid \text{shift } k \text{ in } e \mid \text{reset } e$$

Like call/cc, these operators can be interpreted via a modified CPS translation. Danvy and Filinski call the new translation *extended continuation-passing style* (ECPS):

$$\begin{aligned}C[\![\text{shift } k \text{ in } e]\!] &= (\lambda\kappa.\, C[\![e]\!]\ \text{id})[k \mapsto \lambda x.\, \lambda\kappa'.\, \kappa'\ (\kappa\ x)]\\ C[\![\text{reset } e]\!] &= \lambda\kappa.\, \kappa\ (C[\![e]\!]\ \text{id})\end{aligned}$$

The equation for reset e translates e and extracts its final value by applying it to id. This means that any continuation manipulation done by e does not leak outside of the scope of the reset, so we say

that reset "delimits" the continuation. The translation for shift gives e access to its continuation, κ, via the variable k.

Together, shift and reset provide an elegant interface for working with continuations. For example, take the following expression and its evaluation:

$$\begin{aligned}
&1 + \mathsf{reset}\ (10 + \mathsf{shift}\ k\ \mathsf{in}\ k\ (k\ 100)) \Rightarrow \\
&1 + (10 + (10 + 100)) \Rightarrow \\
&121
\end{aligned}$$

The continuation, $k = \lambda x.\, 10 + x$, is captured by the shift operator, since "$10 + \mathsf{shift}\ k\ \mathsf{in}\ \ldots$" is within a context delimited by reset. The outer "1 +" is not captured because it is outside of the reset. The function k is applied twice to 100, resulting in 120, and then we add 1 to get 121.

2.2 A Denotational Semantics

Danvy and Filinski point out that the ECPS translation, with rules for shift and reset, is no longer truly in continuation-passing style. In particular, the rule for shift means that the resulting program might not enforce strict call-by-value semantics, since the translation of shift introduces a redex $\kappa'\ (\kappa\ x)$. This could be solved by translating the result a second time, but we can use this as an opportunity to take a step back and look at another way of defining the semantics of our calculus with shift and reset.

In addition to the ECPS translation, Danvy and Filinski give a denotational translation. Whereas the ECPS translation is purely syntactic, this transformation is *semantic* and maps every program to a mathematical object in a given *domain*. In this case the domain contains Booleans (since we have true and false as base values) and computable functions. This is common practice in programming languages literature and hearkens back to the origin of continuations [15]. Assuming Ans is a suitable domain of final answers, we define:

$$\begin{array}{lcl}
\rho \in \mathsf{Env} & = & \mathsf{Var} \rightharpoonup \mathsf{Val} \\
\gamma \in \mathsf{MCont} & = & \mathsf{Val} \to \mathsf{Ans} \\
\kappa \in \mathsf{Cont} & = & \mathsf{Val} \to \mathsf{MCont} \to \mathsf{Ans} \\
\mathcal{E} & : & \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Cont} \to \mathsf{MCont} \to \mathsf{Ans}
\end{array}$$

The variable environment, ρ, maps variables to domain values, γ is a "meta"-continuation that captures the continuation *outside* of the closest enclosing reset, and κ is the continuation that we are used to (inside of the closest reset). The denotational semantics is given by the equations:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\ \rho\ \kappa\ \gamma &= \kappa\ (\rho[x])\ \gamma \\
\mathcal{E}[\![\lambda x.\, e]\!]\ \rho\ \kappa\ \gamma &= \kappa\ (\lambda v.\, \mathcal{E}[\![e]\!]\ (\rho[x \mapsto v]))\ \gamma \\
\mathcal{E}[\![e_1\ e_2]\!]\ \rho\ \kappa\ \gamma &= \mathcal{E}[\![e_1]\!]\ \rho\ (\lambda f.\, \mathcal{E}[\![e_2]\!]\ \rho\ (\lambda x.\, f\ x\ \kappa))\ \gamma \\
\mathcal{E}[\![\mathsf{true}]\!]\ \rho\ \kappa\ \gamma &= \kappa\ \mathsf{true}\ \gamma \\
\mathcal{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\ \rho\ \kappa\ \gamma &= \mathcal{E}[\![e_1]\!]\ \rho\ (\lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ \mathcal{E}[\![e_2]\!]\ \rho\ \kappa\ \mathsf{else}\ \mathcal{E}[\![e_3]\!]\ \rho\ \kappa)\ \gamma \\
\mathcal{E}[\![\mathsf{shift}\ k\ \mathsf{in}\ e]\!]\ \rho\ \kappa\ \gamma &= \mathcal{E}[\![e]\!]\ (\rho[k \mapsto \lambda x.\, \lambda\kappa'.\, \lambda\gamma'.\, \kappa\ x\ (\lambda w.\, \kappa'\ w\ \gamma')])\ (\lambda x.\, \lambda\gamma''.\, \gamma''\ x)\ \gamma \\
\mathcal{E}[\![\mathsf{reset}\ e]\!]\ \rho\ \kappa\ \gamma &= \mathcal{E}[\![e]\!]\ \rho\ (\lambda x.\, \lambda\gamma'.\, \gamma'\ x)\ (\lambda x.\, \kappa\ x\ \gamma)
\end{aligned}$$

It is safe to ignore the γ arguments for all rules other than the ones for shift and reset, since γ can be η-reduced away (since $\lambda x.\, f\ x$ is equivalent to f). Without γ, the first five equations look a lot like the standard CPS translation, just lifted to a semantic domain. The meta-continuations are used in the shift and reset rules to capture the outer continuation as a computable function. Remember that the constructions on the right-hand side are part of a meta-language and represent mathematical objects rather than concrete syntax. Our continuations are embedded in the meta-theory.

All of this is important for theoretical work on this language, but the syntactic ECPS transformation is more useful for concrete implementations. Next, we return to the ECPS translation and explore ways that it can be made more efficient.

2.3 Metacircular Interpreters

A *metacircular interpreter* is a powerful tool for expressing a language's semantics [12]. In the case of our shift–reset language, the metacircular interpreter looks a lot like the ECPS translation, but it does a bit more work by translating constructs an executable meta-language rather than generating syntax directly.

The meta-language for this interpreter, is the same lambda calculus we have been working with, but without shift or reset. We use a simple *quoting* mechanism to specify which parts of the output are executable code and which are concrete syntax. The code in black is the executable interpreter, while the ⌜quoted⌝ code is concrete syntax output. We use ⦇unquote brackets⦈ to splice computations into quoted segments. The translation is defined as:

$$
\begin{aligned}
C'[\![x]\!] &= \lambda\kappa.\, \kappa\ x \\
C'[\![\lambda x.\, e]\!] &= \lambda\kappa.\, \kappa\ \ulcorner \lambda(\!|x|\!).\, \lambda k.\, (\!|C'[\![e]\!]\ (\lambda a.\, \ulcorner k\ (\!|a|\!) \urcorner)|\!) \urcorner \\
C'[\![e_1\ e_2]\!] &= \lambda\kappa.\, C'[\![e_1]\!]\ (\lambda f.\, C'[\![e_2]\!]\ (\lambda x.\, \ulcorner (\!|f|\!)\ (\!|x|\!)\ (\lambda t.\, (\!|\kappa\ \ulcorner t \urcorner|\!)) \urcorner)) \\
C'[\![\mathsf{true}]\!] &= \lambda\kappa.\, \kappa\ \mathsf{true} \\
C'[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] &= \lambda\kappa.\, C'[\![e_1]\!]\ (\ulcorner \lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ (\!|C'[\![e_2]\!]\ \kappa|\!)\ \mathsf{else}\ (\!|C'[\![e_3]\!]\ \kappa|\!) \urcorner) \\
C'[\![\mathsf{shift}\ k\ \mathsf{in}\ e]\!] &= (\lambda\kappa.\, C'[\![e]\!]\ \mathrm{id})[k \mapsto \ulcorner \lambda x.\, \lambda\kappa'.\, \kappa'\ (\!|\kappa\ \ulcorner x \urcorner|\!) \urcorner] \\
C'[\![\mathsf{reset}\ e]\!] &= \lambda\kappa.\, \kappa\ (C'[\![e]\!]\ \mathrm{id})
\end{aligned}
$$

and the final ECPS result is given by applying $C'[\![e]\!]$ to the identity function.

Like in to the denotational semantics, the continuations are all meta-functions; there are no κs in the resulting programs. The difference is that the meta-functions are executed to produce concrete syntax, rather than interpreted as abstract functions. For example, the rule for if-statements will produce a concrete term of the form $\lambda b.\, \mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_$, rather than a mathematical object. Representing continuations at the meta-level drastically reduces the size of the resulting programs—there are no unnecessary redexes in the output.

It turns out that we can do even better if we include shift and reset in our meta-language. This final ECPS translation,

$$
\begin{aligned}
C''[\![x]\!] &= x \\
C''[\![\lambda x.\, e]\!] &= \ulcorner \lambda(\!|x|\!).\, \lambda\kappa.\, (\!|\mathsf{reset}\ (\ulcorner \kappa\ (\!|C''[\![e]\!]|\!) \urcorner)|\!) \urcorner \\
C''[\![e_1\ e_2]\!] &= \mathsf{shift}\ \kappa\ \mathsf{in}\ \ulcorner (\!|C''[\![e_1]\!]|\!)\ (\!|C''[\![e_2]\!]|\!)\ (\lambda t.\, (\!|\kappa\ \ulcorner t \urcorner|\!)) \urcorner \\
C''[\![\mathsf{true}]\!] &= \mathsf{true} \\
C''[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] &= \mathsf{shift}\ \kappa\ \mathsf{in} \\
&\qquad \ulcorner \mathsf{if}\ (\!|C''[\![e_1]\!]|\!)\ \mathsf{then}\ (\!|\mathsf{reset}\ (\kappa\ C''[\![e_2]\!])|\!)\ \mathsf{else}\ (\!|\mathsf{reset}\ (\kappa\ C''[\![e_3]\!])|\!) \urcorner \\
C''[\![\mathsf{shift}\ k\ \mathsf{in}\ e]\!] &= \mathsf{shift}\ \kappa\ \mathsf{in}\ (\mathsf{reset}\ C''[\![e]\!])[k \mapsto \ulcorner \lambda x.\, \lambda\kappa'.\, \kappa'\ (\!|\kappa\ \ulcorner x \urcorner|\!) \urcorner] \\
C''[\![\mathsf{reset}\ e]\!] &= \mathsf{reset}\ C''[\![e]\!]
\end{aligned}
$$

also avoids unnecessary η-expansion (i.e., it always produces f instead of $\lambda x.\, f\ x$), resulting in a pleasingly compact final representation.

This final interpreter is essentially written in the same language that it interprets—this is why metacircular interpreters are called "meta"—and that poses a bootstrapping problem. Luckily, we

already have both the ECPS translation and our denotational semantics as descriptions of this meta-language. The efficient interpreter can be written in this meta-language and then that interpreter can itself be ECPS transformed into a standard lambda calculus.

2.4 Use Case: Nondeterministic Programming

We can understand this machinery better by examining a use-case for delimited continuations that Danvy and Filinski mention in their paper: nondeterministic computations.

We start by defining operators fail and flip, which form the basis of nondeterministic computation:

$$\begin{aligned}\text{fail}\ () &= \text{shift}\ k\ \text{in}\ \text{“failure”}\\ \text{flip}\ () &= \text{shift}\ k\ \text{in}\ (k\ \text{true};\ k\ \text{false};\ \text{fail}\ ())\end{aligned}$$

We can think of flip as flipping a coin and fail as signaling a dead-end outcome.

Concretely the fail operation acts as an exception. It throws away the continuation and returns the string “failure”. The flip operation calls its continuation on *both* true and false, which simulates a nondeterministic choice between the two options. This works because the result of the nondeterministic computation is actually a sequence of values—all of the possible outcomes. When we call flip, the possible options are true and false, so we continue with both. If later in the computation one of those values is undesirable, we can call fail.

The choice operation builds on flip to simulate a nondeterministic choice of positive integers less than a given value.

$$\text{choice}\ n = \text{if}\ n < 1\ \text{then}\ \text{fail}\ ()\ \text{else if}\ \text{flip}\ ()\ \text{then}\ \text{choice}\ (n-1)\ \text{else}\ n$$

Calling choice 2 will continue with 2, then 1, and then “failure”.

Finally, Danvy and Filinski implement the function

$$\begin{aligned}&\text{triple}\ n\ s =\\ &\qquad \text{let}\ i = \text{choice}\ n\ \text{in}\\ &\qquad \text{let}\ j = \text{choice}\ (i-1)\ \text{in}\\ &\qquad \text{let}\ k = \text{choice}\ (j-1)\ \text{in}\\ &\qquad \text{if}\ i+j+k = s\ \text{then}\ (i,\ j,\ k)\ \text{else}\ \text{fail}\ (),\end{aligned}$$

which finds all triples of distinct positive integers that sum to a given integer s. The complexity is pushed into choice, so this function is nice and readable. Evaluating the expression

$$\text{reset}\ (\text{print}\ (\text{triple}\ 9\ 15))$$

prints all triples of integers up to 9 that sum to 15.

The triple example is mostly a toy based on an early benchmark, but these primitives can be practically useful. In their paper, Danvy and Filinski show an evaluator for *nondeterministic finite automata* (NFAs), which is straightforward to implement thanks to operators like flip and fail. The implementation is not especially instructive, so I do not repeat it here.

We return to continuations later on, but for now we can take a break from continuations and look at monads.

3 *COMPREHENDING MONADS*

As mentioned in Section 1, monads are as popular as they are poorly understood. They were originally introduced to the programming languages literature by Moggi [10], who was working in *category theory*. Luckily, monads can be understood without learning a new branch of mathematics, and *Comprehending Monads* by Wadler [16] is a great place to start.

A monad is essentially just a data structure (e.g., List or Maybe) with certain associated operations. First, a monad must have an operation map that applies a function to the "elements" of the data structure. For example,

$$\mathsf{map}^{\mathsf{List}}\ f\ \overline{x}$$

applies f to all elements of $\overline{x}$ and

$$\mathsf{map}^{\mathsf{Maybe}}\ f\ \overline{x}$$

applies f to the value contained in $\overline{x}$, if one exists. (I adopt Wadler's convention of writing monadic values with a bar over the variable name.) The map operation must obey two laws:

$$\mathsf{map}\ \mathsf{id} = \mathsf{id}$$
$$\mathsf{map}\ (g \circ f) = \mathsf{map}\ g \circ \mathsf{map}\ f$$

The first law says that mapping the identity function over a structure does nothing—this is necessary to make sure that map does not change the structure itself, only the elements in the structure. The second law says that map is well-behaved with respect to function composition.

A data structure with a valid map operation is called a *functor*. In order for a functor to be a monad, it needs two more operations: unit and join. Applying unit to a value injects that value into the monad structure. In the case of List,

$$\mathsf{unit}^{\mathsf{List}}\ x = \mathsf{singleton}\ x = [x],$$

which is the simplest way to inject a value into a list. Likewise,

$$\mathsf{unit}^{\mathsf{Maybe}}\ x = \mathsf{Just}\ x.$$

The join operation works on "doubled up" instances of the monad. It takes a doubled value like

$$[[1,2],[3,4]] \quad \text{or} \quad \mathsf{Just}\ (\mathsf{Just}\ 5)$$

and flattens it down to a single application of the monad like

$$[1,2,3,4] \quad \text{or} \quad \mathsf{Just}\ 5.$$

To be precise:

$$\begin{aligned} \mathsf{join}^{\mathsf{List}}\ \overline{\overline{x}} &= \mathsf{flatten}\ \overline{\overline{x}} \\ \mathsf{join}^{\mathsf{Maybe}}\ \overline{\overline{x}} &= \mathsf{case}\ \overline{\overline{x}}\ \mathsf{of}\ \{\ \mathsf{Just}\ (\mathsf{Just}\ x) \to \mathsf{Just}\ x;\ \_ \to \mathsf{Nothing}\ \} \end{aligned}$$

These operations must obey certain laws in order for the structure to truly be a monad:

$$\mathsf{map}\ f \circ \mathsf{unit} = \mathsf{unit} \circ f \tag{1}$$
$$\mathsf{map}\ f \circ \mathsf{join} = \mathsf{join} \circ \mathsf{map}\ (\mathsf{map}\ f) \tag{2}$$
$$\mathsf{join} \circ \mathsf{unit} = \mathsf{id} \tag{3}$$
$$\mathsf{join} \circ \mathsf{map}\ \mathsf{unit} = \mathsf{id} \tag{4}$$
$$\mathsf{join} \circ \mathsf{join} = \mathsf{join} \circ \mathsf{map}\ \mathsf{join} \tag{5}$$

Law (1) and (2) enforce that unit and join are well-behaved with respect to map. In particular, (1) is a strong statement that says that unit cannot *do* very much—it must commute with an arbitrary function f (modulo the monad structure). Law (3) says that applying unit to a monadic value and then join-ing is a no-op. Law (4) says the same as (3) except we apply unit *under* the monad using map. Finally, law (5) says that for a triple-stacked monad (e.g., $[[[1]]]$), either way of flattening it to a single monad (e.g., $[1]$) is the same.

Importantly, the monad laws require that unit and join are the *simplest* ways to do their respective tasks. For example, suppose we tried to define:

$$\mathsf{unit}^{\mathsf{List}}\ x = [x,\ x,\ x,\ x,\ x]$$

Technically this does inject a value into a list, but it is not the simplest way to do so and

$$(\mathsf{join}^{\mathsf{List}} \circ \mathsf{unit}^{\mathsf{List}})\ [3] = [3, 3, 3, 3, 3] \neq \mathsf{id}\ [3].$$

All of this background on monads and their operations is important, but in practice using these operations directly does not make for particularly elegant code. In the next section, I explore Wadler's use of *comprehensions* to make programming with monads much more intuitive.

3.1 Comprehensions

Comprehending Monads does more than just explain monads. As the name cleverly suggests, the paper develops a notation for monadic computations derived from list comprehensions.

List comprehensions are based on set-builder notation (e.g. $\{n + 1 \mid n \in \mathbb{N}\}$) and were first introduced to programming by the SETL language in the 1960s [14]. Since then, list comprehensions have become ubiquitous, appearing in many modern languages including Racket, Python, and even C++ (with some abuse of operator overloading). Comprehensions are a convenient way of building lists from other lists, for example a construction like

$$[(x, y) \mid x \leftarrow \overline{x};\ y \leftarrow \overline{y}]^{\mathsf{List}}$$

computes the Cartesian product of $\overline{x}$ and $\overline{y}$. The analogous set-builder construction looks like

$$\{(x, y) \mid x \in X \wedge y \in Y\}.$$

List comprehensions have the form $[e \mid q]$, where q is called a *qualifier*. Wadler defined the syntax of qualifiers as:

$$q ::= \Lambda \mid x \leftarrow e \mid (q_1;\ q_2)$$

Qualifiers can be empty (denoted Λ), a binder for a variable x, or a composition of two qualifiers. This notation can be a bit heavy, so Wadler usually writes "$[e]$" instead of "$[e \mid \Lambda]$," and he proves that qualifier composition is associative so he can write "$[e \mid q_1;\ q_2]$" instead of $[e \mid (q_1;\ q_2)]$.

Comprehending Monads centers around the observation that lists are not the only structure for which comprehensions make sense. We can characterize behavior of list comprehensions with following rules:

$$\begin{aligned}
[e \mid \Lambda] &= \mathsf{singleton}\ e \\
[e_1 \mid x \leftarrow e_2] &= \mathsf{map}\ (\lambda x.\, e_1)\ e_2 \\
[e \mid (q_1;\ q_2)] &= \mathsf{flatten}\ [[e \mid q_2] \mid q_1]
\end{aligned}$$

Notice that singleton is $\mathsf{unit}^{\mathsf{List}}$ and flatten is $\mathsf{join}^{\mathsf{List}}$. Would using unit and join for other monads give a reasonable way to re-interpret comprehensions?

Yes! Wadler defines a lambda calculus with generic *monad comprehension* syntax:

$$\begin{aligned}
e &::= \cdots \mid [e \mid q]^{\mathsf{M}} \\
q &::= \Lambda \mid x \leftarrow e \mid (q_1;\ q_2)
\end{aligned}$$

For any monad M the monad comprehension behaves according to the following rules:

$$\begin{aligned}
[e \mid \Lambda]^{\mathsf{M}} &= \mathsf{unit}^{\mathsf{M}}\ e \\
[e_1 \mid x \leftarrow e_2]^{\mathsf{M}} &= \mathsf{map}^{\mathsf{M}}\ (\lambda x.\, e_1)\ e_2 \\
[e \mid (q_1;\ q_2)]^{\mathsf{M}} &= \mathsf{join}^{\mathsf{M}}\ [[e \mid q_2]^{\mathsf{M}} \mid q_1]^{\mathsf{M}}
\end{aligned}$$

The empty qualifier simply injects e into the monad using unit. The binding qualifier defines a function: e_2 produces some monadic value, and the result of the comprehension is $\lambda x.\, e_1$ mapped over that value. Finally, qualifier composition translates to a nested comprehension which is

flattened using join. Note that while Wadler opts to extend the language with comprehensions, they could instead be implemented as syntactic sugar—in other words, they can be implemented as surface syntax and omitted from the formal language.

Comprehensions can be used for computations like the Maybe comprehension from Section 1,

$$[i/j \mid i \leftarrow [84]^{\text{Maybe}};\ j \leftarrow \text{if } n = 0 \text{ then Nothing else } [n]^{\text{Maybe}}]^{\text{Maybe}}$$

which cleanly handles the fact that the expression binding j might fail, without introducing significant syntactic overhead.

3.2 Examples of Monads

List and Maybe are far from the only monads, and while the formal definitions are important, the best way to understand monads is by example. Wadler presents a whole zoo of monads in his paper; exploring those structures provides valuable intuition for how monads work and what they can do.

For each of these monads, I define the monad data structure (with a fairly standard syntax, similar to Haskell) and the three operations, map, unit, and join. I assume that we have case-statements in our language for destructuring these data structures.

3.2.1 List and Maybe. **The Running Examples**

$$\begin{array}{lcl}
\text{data List } \alpha & = & \text{Nil} \mid \text{Cons } \alpha\ (\text{List } \alpha) \\
\text{map}^{\text{List}}\ f\ \overline{x} & = & \text{case } \overline{x} \text{ of } \{\ \text{Nil} \to \text{Nil};\ \text{Cons } y\ \overline{y} \to \text{Cons } (f\ y)\ (\text{map}^{\text{List}}\ f\ \overline{y})\ \} \\
\text{unit}^{\text{List}}\ x & = & \text{singleton } x \\
\text{join}^{\text{List}}\ \overline{\overline{x}} & = & \text{flatten } \overline{\overline{x}} \\
 & & \\
\text{data Maybe } \alpha & = & \text{Nothing} \mid \text{Just } \alpha \\
\text{map}^{\text{Maybe}}\ f\ \overline{x} & = & \text{case } \overline{x} \text{ of } \{\ \text{Nothing} \to \text{Nothing};\ \text{Just } y \to \text{Just } (f\ y)\ \} \\
\text{unit}^{\text{Maybe}}\ x & = & \text{Just } x \\
\text{join}^{\text{Maybe}}\ \overline{\overline{x}} & = & \text{case } \overline{\overline{x}} \text{ of } \{\ \text{Just } (\text{Just } x) \to \text{Just } x;\ _ \to \text{Nothing}\ \}
\end{array}$$

We have already seen the definitions for List and Maybe: the List monad coincides exactly with list comprehensions, and the Maybe monad can be used to sequence computations that might fail (returning Nothing).

3.2.2 Identity and Strictness. **The Degenerate Cases**

$$\begin{array}{lcl}
\text{data Id } \alpha & = & \alpha \\
\text{map}^{\text{Id}}\ f\ \overline{x} & = & f\ \overline{x} \\
\text{unit}^{\text{Id}}\ x & = & x \\
\text{join}^{\text{Id}}\ \overline{\overline{x}} & = & \overline{\overline{x}} \\
 & & \\
\text{data Str } \alpha & = & \alpha \\
\text{map}^{\text{Str}}\ f\ \overline{x} & = & \text{if } \overline{x} \neq \bot \text{ then } f\ \overline{x} \text{ else } \bot \\
\text{unit}^{\text{Str}}\ x & = & x \\
\text{join}^{\text{Str}}\ \overline{\overline{x}} & = & \overline{\overline{x}}
\end{array}$$

List and Maybe are relatively simple monads, but there are a couple that are even simpler. The most basic monad is just an identity—it does nothing. This monad is almost not worth mentioning, but it does have one cute use-case: comprehension syntax for the identity monad can be used instead of let binding. We can write

$$[e_3 \mid x \leftarrow e_1;\ y \leftarrow e_2]^{\text{Id}}$$

instead of

$$\text{let } x = e_1 \text{ in let } y = e_2 \text{ in } e_3.$$

There is no real reason to prefer this syntax, so I will continue to use let, but it is a fun observation.

A slight variation on the identity monad, the *strictness* monad, does actually have some interesting uses. Assuming a call-by-name base language, the strictness monad provides control over evaluation order by altering the definition of map to force the evaluation of the argument $\overline{x}$. The expression

$$[e_3 \mid x \leftarrow e_1;\ y \leftarrow e_2]^{\text{Str}}$$

works like

$$\text{let } !x = e_1 \text{ in let } !y = e_2 \text{ in } e_3$$

where let $!x = e_1$ in e_2 forces the evaluation of e_1 before assigning to x. Str comprehensions give access to call-by-value semantics as needed, without changing the underlying language.

3.2.3 State. **Pure "Imperative" Programming**

$$\begin{array}{lcl}
\text{data State } \alpha & = & \sigma \to (\alpha, \sigma) \\
\text{map}^{\text{State}}\ f\ \overline{x} & = & \lambda s.\ \text{let } (x, s') = \overline{x}\ s \text{ in } (f\ x, s') \\
\text{unit}^{\text{State}}\ x & = & \lambda s.\ (x, s) \\
\text{join}^{\text{State}}\ \overline{\overline{x}} & = & \lambda s.\ \text{let } (\overline{x}, s') = \overline{\overline{x}}\ s \text{ in } \overline{x}\ s'
\end{array}$$

The State monad enables code that *looks* stateful, even though there are no actual references under the hood. Assuming a fixed type σ of states, the type $\sigma \to (\alpha, \sigma)$ says that each stateful computation takes a state as input and produces a *result* of type α along with a new state.

Since State is such an important and interesting monad, we should look at the operations in a bit of detail. As usual, unit is the simplest; it simply injects a value into a stateful computation. The state stays the same, and the result is the argument passed to unit. The map operation takes a computation, runs it, and then applies a function to the result value. Finally, join runs a computation which results in *another* computation, and then it runs that computation to get a final result. These operations together mean that state comprehensions act like imperative code, with state handled implicitly.

As with many monads, interesting programs in the state-monad make use of some auxiliary operations:

$$\begin{array}{lcl}
\text{get} & = & \lambda s.\ (s, s) \\
\text{put } s' & = & \lambda s.\ ((), s')
\end{array}$$

The get operation accesses the state, and the put operation sets the state. These can be used to write a program like

$$p = [k \mid i \leftarrow \text{get};\ \_ \leftarrow \text{put } (i + 3);\ j \leftarrow \text{get};\ \_ \leftarrow \text{put } (j * 7);\ k \leftarrow \text{get}]^{\text{State}},$$

which simulates a simple stateful computation. We can call p 3 (choosing 3 as the initial state) to get the pair $(42, 42)$. If we expand the comprehension syntax and monad operation definitions,

$$\begin{array}{l}
\lambda s_0.\ \text{let } (i, s_1) = \text{get } s_0 \text{ in} \\
\quad \text{let } ((), s_2) = \text{put } (i + 3)\ s_1 \text{ in} \\
\quad \text{let } (j, s_3) = \text{get } s_2 \text{ in} \\
\quad \text{let } ((), s_4) = \text{put } (j * 7)\ s_3 \text{ in} \\
\quad \text{let } (k, s_5) = \text{get } s_4 \text{ in} \\
\quad (k, s_5),
\end{array}$$

we can see that the state parameters s_0, s_1, etc. are being passed from one expression to the next, tracking the state as it evolves.

3.2.4 Reader. **AKA "State Reader"**

$$\begin{array}{lcl} \mathsf{data\ Reader}\ \alpha & = & \rho \to \alpha \\ \mathsf{map}^{\mathsf{Reader}}\ f\ \overline{x} & = & \lambda r.\, f\ (\overline{x}\ r) \\ \mathsf{unit}^{\mathsf{Reader}}\ x & = & \lambda r.\, x \\ \mathsf{join}^{\mathsf{Reader}}\ \overline{\overline{x}} & = & \lambda r.\, (\overline{\overline{x}}\ r)\ r \end{array}$$

The Reader monad is a simplification of the State monad that does not allow the state to be modified. Despite its relative simplicity, the Reader monad is quite powerful. After defining the auxiliary operation,

$$\mathsf{ask} \quad = \quad \lambda r.\, r,$$

Reader can be used to keep track of configuration values, environment information, and any other static value that would otherwise be cumbersome to pass around explicitly. Readers are also safer and more flexible than global variables or constants, since they are implemented as functions and thus have a clearly delimited scope.

3.2.5 Nondeterminism. **The (Power) Set Monad**

$$\begin{array}{lcl} \mathsf{data\ ND}\ \alpha & = & \mathcal{P}(\alpha) \\ \mathsf{map}^{\mathsf{ND}}\ f\ \overline{x} & = & \{f\ x \mid x \in \overline{x}\} \\ \mathsf{unit}^{\mathsf{ND}}\ x & = & \{x\} \\ \mathsf{join}^{\mathsf{ND}}\ \overline{\overline{x}} & = & \bigcup \overline{\overline{x}} \end{array}$$

The nondeterminism monad enables direct-style programming of nondeterministic algorithms, much like the triple example from Section 2.4. A nondeterministic value is really a set of values, corresponding to each of the possible outcomes of a nondeterministic computation. We define the same operators that we saw before:

$$\begin{array}{lcl} \mathsf{fail} & = & \varnothing \\ \mathsf{flip} & = & \{\mathsf{true}, \mathsf{false}\}, \end{array}$$

We can fail by returning the empty set—there are no possible outcomes—and we can also flip between true and false as expected.

The monad operations keep track of all possible outcomes at a given point in the program, but programs can be written as if there is only one path.

3.2.6 Parser. **Top-Down Parser Combinators**

$$\begin{array}{lcl} \mathsf{data\ Parser}\ \alpha & = & \mathsf{String} \to \mathsf{List}\ (\alpha, \mathsf{String}) \\ \mathsf{map}^{\mathsf{Parser}}\ f\ \overline{x} & = & \lambda i.\, [(f\ x,\ i') \mid (x,\ i') \leftarrow \overline{x}\ i]^{\mathsf{List}} \\ \mathsf{unit}^{\mathsf{Parser}}\ x & = & \lambda i.\, [(x, i)]^{\mathsf{List}} \\ \mathsf{join}^{\mathsf{Parser}}\ \overline{\overline{x}} & = & \lambda i.\, [(x,\ i'') \mid (\overline{x},\ i') \leftarrow \overline{\overline{x}}\ i;\ (x,\ i'') \leftarrow \overline{x}\ i']^{\mathsf{List}} \end{array}$$

Finally, here is a more complicated monad: Parser. Parsers can be seen as a combination of two monads: List and State. Intuitively, the String argument is the parser input, and the result is a list of potential parses, each of which is made up of a result of type α and the String that remains after parsing.

There are a number of auxiliary operations that are useful for Parser (often called *parser combinators*), including satisfy which parses a character matching a given predicate and many

which applies a parser multiple times. An amazing amount of research has been done on the Parser monad and parser combinators [6, 8], and it is still an active area [18].

3.3 Other Formulations of Monads

Before moving on, I want to address some other formulations of monads that have become more popular since *Comprehending Monads* was published.

A monad's join operation can be used to define special kind of function composition called *Kleisli composition*. Given a monad M and two functions $f : \beta \to \mathsf{M}\ \gamma$ and $g : \alpha \to \mathsf{M}\ \beta$, we can write a function

$$f \circ_{\mathsf{M}} g = \mathsf{join}^{\mathsf{M}} \circ \mathsf{map}^{\mathsf{M}}\ f \circ g$$

that composes them together while properly keeping track of M. This is useful for programming because it means that monadic operations like f and g can be built up individually and composed together to build larger programs. A very similar construction can also yield the bind (or »=) definition of monads that is more common in modern programming languages.

The bind definition of monads admits syntactic sugar called *do–notation*, which is popular in Haskell. Do–notation is actually very similar to monad comprehension syntax:

$$\mathsf{do}\ \{\ i \leftarrow \mathsf{unit}^{\mathsf{Maybe}}\ 84;\ j \leftarrow (\mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{Nothing}\ \mathsf{else}\ \mathsf{unit}^{\mathsf{Maybe}}\ n);\ \mathsf{unit}^{\mathsf{Maybe}}\ i/j\ \}$$

The distinctions between comprehension syntax and do–notation are mostly cosmetic. When using do–notation, programmers write "$\mathsf{unit}^{\mathsf{Maybe}}\ e$" (or more idiomatically "return e") instead of "$[e]^{\mathsf{M}}$." Also rather than put a return value at the front of the comprehension, do–blocks evaluate to their final expression. For the remainder of this paper I stick to comprehension syntax, but programming-focused monad examples usually use something closer to do–notation.

Next, we examine examples of monads that are defined using delimited continuations.

4 COMMON DESIGN PATTERNS

The monadic design patterns that Wadler presents are also a great showcase for delimited continuations. We have already seen delimited continuations implement exceptions (Maybe) and nondeterminism (ND), but it turns out that they can be used to implement all of the monads from the previous section.

This presentation was inspired by a blog post by Xia [19], which shows how to implement a variety of monads using continuation-passing. Following Xia's definitions, I present each pattern by defining (1) the primitive operations for the pattern and (2) a run function that delimits the scope of the computation using reset.

4.0.1 Maybe. **See Section 3.2.1**

$$\begin{array}{lcl} \mathsf{abort} & = & \mathsf{shift}\ k\ \mathsf{in}\ \mathsf{Nothing} \\ \mathsf{run}^{\mathsf{Maybe}}\ c & = & \mathsf{reset}\ (\mathsf{Just}\ c) \end{array}$$

We have already seen exceptions in the contexts of both continuations and monads already, although I have been imprecise when "throwing an exception" using shift. This construction faithfully simulates the Maybe monad, since its computations either fail and return Nothing or succeed with a value v and return Just v.

4.0.2 Identity and Strictness. **See Section 3.2.2**

$$\begin{array}{lcl} \mathsf{noop}\ e & = & \mathsf{shift}\ k\ \mathsf{in}\ k\ e \\ \mathsf{run}^{\mathsf{Id}}\ c & = & \mathsf{reset}\ c \\ \\ \mathsf{force}\ e & = & \mathsf{shift}\ k\ \mathsf{in}\ k\ (\mathsf{if}\ e \neq \bot\ \mathsf{then}\ e\ \mathsf{else}\ \bot) \\ \mathsf{run}^{\mathsf{St}}\ c & = & \mathsf{reset}\ c \end{array}$$

Identity can be encoded without using continuations at all, of course, but this presentation is instructive. Here the noop operation shows how to do nothing using delimited continuations—just call the continuation. All run needs to do is delimit the continuation scope.

Just like with monads, we can also encode strictness (provided our source language is non-strict). We do the same trick as before, forcing e and then calling the continuation.

4.0.3 State. **See Section 3.2.3**

$$\begin{array}{lcl} \mathsf{get} & = & \mathsf{shift}\ k\ \mathsf{in}\ \lambda s.\, k\ s\ s \\ \mathsf{put}\ s' & = & \mathsf{shift}\ k\ \mathsf{in}\ \lambda s.\, k\ ()\ s' \\ \mathsf{run}^{\mathsf{State}}\ c\ i & = & \mathsf{reset}\ ((\lambda v.\, \lambda s.\, v)\ c)\ i \end{array}$$

As with the state monad, encoding state with continuations is about capturing state in function parameters and return values. Our continuations take two parameters: the first is the computation result, and the second is the new state. The get operation takes in the current state and continues with that state as both the result and the new state. The put operation ignores the current state and continues with unit as the result and s' as the new state. The run function takes an initial state and passes it in *outside* the continuation scope.

As with the monadic version, these operations can be used to simulate stateful code. For example, this program, which is essentially the same program from Section 3.2.3,

$$\begin{array}{l} \mathsf{run}^{\mathsf{State}}\ (\mathsf{let}\ i = \mathsf{get}\ \mathsf{in} \\ \qquad \mathsf{put}\ (i + 3); \\ \qquad \mathsf{let}\ j = \mathsf{get}\ \mathsf{in} \\ \qquad \mathsf{put}\ (j * 7); \\ \qquad \mathsf{get})\ 3 \end{array}$$

returns 42 as expected. In this case the control flow is a bit more complicated, but the intuition is the same as for the monadic version.

4.0.4 Reader. **See Section 3.2.4**

$$\begin{array}{lcl} \mathsf{ask} & = & \mathsf{get} \\ \mathsf{run}^{\mathsf{Reader}} & = & \mathsf{run}^{\mathsf{State}} \end{array}$$

Reader is strictly simpler than State, but we can use the same definitions anyway. We just rename get to ask and don't provide a put operation.

4.0.5 Nondeterminism. **See Section 3.2.5**

$$\begin{array}{lcl} \mathsf{fail} & = & \mathsf{shift}\ k\ \mathsf{in}\ k\ \varnothing \\ \mathsf{flip} & = & \mathsf{shift}\ k\ \mathsf{in}\ k\ \mathsf{true} \cup k\ \mathsf{false} \\ \mathsf{run}^{\mathsf{ND}}\ c & = & \mathsf{reset}\ ((\lambda v.\, \{v\})\ c) \end{array}$$

We already saw one way of encoding nondeterminism using continuations, but here is another one that is a bit closer to the monadic presentation. We again make use of sets to define the primitive operations fail and flip: Our implementation of run lifts computations into singleton sets, and our operators call the continuation on whatever values should be considered—either nothing in the case of fail or both true and false in the case of flip.

4.1 Other Monad Implementations

A couple of Wadler's monad examples are missing here, but not for any fundamental reason. List can be implemented using delimited continuations without issues, but there are a number of subtly different ways to do it and the technicalities are not informative. Parsers can also be done using shift and reset, but the construction is complex and, again, not informative. It would seem that anything that can be done with monads can also be done with delimited continuations.

It turns out that with enough effort, this is roughly true! A few years after *Abstracting Control*, Filinski [4] published another paper showing that delimited continuations can represent all *pure* monads (roughly those whose map, join, and bind operations can be implemented in our base lambda calculus). While the constructions in Section 4 are ad-hoc and dependent on the particular monad in question, Filinski managed to give a translation that represents monadic patterns with continuations once-and-for-all. Still, the ad-hoc connections are valuable: they provide more granular intuition than Filinski's presentation, and they are all entirely pure, while Filinski's formulation relies on a single reference cell.

5 THE CONTINUATION MONAD

Up to this point we have viewed delimited continuations and monads in parallel, bouncing back and forth between the abstractions and showing how they compare. But there is one final insight from Wadler's paper that I skipped over, and it serves as a bridge between continuations and monads.

Towards the end of *Comprehending Monads*, Wadler presents the *continuation monad*. Given a result type, ρ, the continuation monad is defined as:

$$\begin{array}{lcl} \mathsf{data}\ \mathsf{Cont}\ \alpha & = & (\alpha \to \rho) \to \rho \\ \mathsf{map}^{\mathsf{Cont}}\ f\ \overline{x} & = & \lambda\kappa.\, \overline{x}\ (\lambda x.\, \kappa\ (f\ x)) \\ \mathsf{unit}^{\mathsf{Cont}}\ x & = & \lambda\kappa.\, \kappa\ x \\ \mathsf{join}^{\mathsf{Cont}}\ \overline{\overline{x}} & = & \lambda\kappa.\, \overline{x}\ (\lambda\overline{x}.\, \overline{x}\ \kappa) \end{array}$$

Cont α is a computation that take a continuation expecting an input α and producing a result of a fixed type ρ. The map operation composes a function f with the continuation, unit simply continues with a given value, and join untangles nested continuations to enable chained computations.

If this seems complicated, it is, but it builds on the intuition we built up from *Abstracting Control*. Programs written using the continuation monad are automatically in a continuation-passing style. A sequence of operations

$$[(x, y) \mid x \leftarrow \overline{x};\ y \leftarrow \overline{y}]^{\mathsf{Cont}}$$

expands to

$$\lambda\kappa.\, \overline{x}\ (\lambda x.\, \overline{y}\ (\lambda y.\, \kappa\ (x,\ y))),$$

which is precisely the way a tuple of expressions would be evaluated in CPS. The order of evaluation is clear: first $\overline{x}$ is evaluated, then $\overline{y}$ is evaluated, and finally the results are packed into a tuple.

As with many of the monads before, we can define auxiliary operations that interact in useful ways with the monad operations. In particular, we can define our friends shift and reset!

$$\begin{array}{lcl} \mathsf{shift}\ f & = & \lambda\kappa.\, f\ (\lambda x.\, \lambda\kappa'.\, \kappa'\ (\kappa\ x))\ (\lambda x.\, x) \\ \mathsf{reset}\ \overline{x} & = & \lambda\kappa.\, \kappa\ (\overline{x}\ (\lambda x.\, x)) \end{array}$$

These definitions are based closely on the ECPS translation, but whereas in Section 2 we built these operations into the language, now they are simply defined. This means that there is no need to extend any meta-theory—we can interpret these operations in the same lambda calculus that we worked with in Section 3.

In particular, this means that we can use the Cont comprehension to write programs like

$$[x + y \mid x \leftarrow [1];\ y \leftarrow \mathsf{reset}\ [u + v \mid u \leftarrow [10];\ v \leftarrow \mathsf{shift}(\lambda k.\ [b \mid a \leftarrow k\ 100;\ b \leftarrow k\ a])]],$$

which is equivalent to the program

$$1 + \mathsf{reset}\ (10 + \mathsf{shift}\ k\ \mathsf{in}\ k\ (k\ 100))$$

that we saw when discussing *Abstracting Control*. The monad version is a bit more verbose, but, critically, neither version relies on manually managing continuation parameters; they are both written in a direct style.

The continuation monad shows just how closely related continuations and monads are—CPS *is* a monad! But we can actually take this one step further, using a monad embedding defined by Wadler.

5.1 Recovering ECPS

Wadler presents a translation, e^*, that lifts a program of type $\alpha \to \beta$ to one of type $\alpha \to \mathsf{M}\ \beta$ for some monad M. The resulting function takes a value of type α and returns a *computation* of type β. The monad M captures the effects of that computation. Here is the translation in full:

$$\begin{aligned}
x^* &= [x]^{\mathsf{M}} \\
(\lambda x.\ e)^* &= [\lambda x.\ e^*]^{\mathsf{M}} \\
(e_1\ e_2)^* &= [y \mid f \leftarrow e_1^*;\ x \leftarrow e_2^*;\ y \leftarrow f\ x]^{\mathsf{M}} \\
(e_1,\ e_2)^* &= [(x, y) \mid x \leftarrow e_1^*;\ y \leftarrow e_2^*]^{\mathsf{M}} \\
(\mathsf{fst}\ e)^* &= [\mathsf{fst}\ x \mid x \leftarrow e^*]^{\mathsf{M}}
\end{aligned}$$

Each of these rules has a straightforward computational meaning. For example

$$(e_1\ e_2)^* = [y \mid f \leftarrow e_1^*;\ x \leftarrow e_2^*;\ y \leftarrow f\ x]^{\mathsf{M}}$$

says that in order to evaluate a (potentially effectful) application, we evaluate e_1 to f, then evaluate e_2 to x, and then evaluate the application $(f\ x)$, all the while keeping track of any side effects that are produced.

If our source language had an impure effect, for example built-in operations get and put that manipulate a reference cell, we could fix $\mathsf{M} = \mathsf{State}$ and add translations

$$\begin{aligned}
\mathsf{get}^* &= [x \mid x \leftarrow \mathsf{get}]^{\mathsf{State}} \\
(\mathsf{put}\ e)^* &= [u \mid x \leftarrow e^*;\ u \leftarrow \mathsf{put}\ x]^{\mathsf{State}}
\end{aligned}$$

where get and put are defined as in Section 3.2. The result of the translation no longer requires references and is potentially simpler to reason about.

We can apply this transformation to translate programs from Section 2 into programs from Section 3, using target monad $\mathsf{M} = \mathsf{Cont}$. We add rules to translate "shift k in e" into "shift $(\lambda k.\ e)$" and "reset e" into "reset e". Shockingly, we obtain exactly the ECPS translation presented by Danvy

and Filinski!

	Cont Monad	**ECPS**
x^*	$= [x]^{\mathsf{Cont}}$	$= \lambda\kappa.\, \kappa\ x$
$(\lambda x.\, e)^*$	$= [\lambda x.\, e^*]^{\mathsf{Cont}}$	$= \lambda\kappa.\, \kappa\ (\lambda x.\, e^*)$
$(e_1\ e_2)^*$	$= [y \mid f \leftarrow e_1^*;\ x \leftarrow e_2^*;\ y \leftarrow f\ x]^{\mathsf{Cont}}$	$= \lambda\kappa.\, e_1^*\ (\lambda f.\, e_2^*\ (\lambda x.\, f\ x\ \kappa))$
$(e_1,\ e_2)^*$	$= [(x, y) \mid x \leftarrow e_1^*;\ y \leftarrow e_2^*]^{\mathsf{Cont}}$	$= \lambda\kappa.\, e_1^*\ (\lambda x.\, e_2^*\ (\lambda y.\, \kappa(x,\ y)))$
$(\mathsf{fst}\ e)^*$	$= [\mathsf{fst}\ x \mid x \leftarrow e^*]^{\mathsf{Cont}}$	$= \lambda\kappa.\, e^*\ (\lambda x.\, \mathsf{fst}\ x)$
$(\mathsf{shift}\ k\ \mathsf{in}\ e)^*$	$= \mathsf{shift}\ (\lambda k.\, e^*)$	$= (\lambda\kappa.\, e^*\ \mathsf{id})[k \mapsto \lambda x.\, \lambda\kappa'.\, \kappa'\ (\kappa\ x)]$
$\mathsf{reset}\ e^*$	$= \mathsf{reset}\ e^*$	$= \lambda\kappa.\, \kappa\ (e^*\ \mathsf{id})$

This is truly beautiful. It means that the continuation monad entirely captures the semantics of a language with first-class continuation operators. Even though Danvy and Filinski give their language no fewer than four semantic interpretations (including ECPS, a denotational semantics, and two meta-circular interpreters) Wadler managed to find one more way of looking at continuation semantics.

(Of course, Wadler did not present exactly this result. He observed that the call-by-name translation results in the unusual call-by-name version of the CPS translation, but since shift and reset did not exist, he did not complete the ECPS picture. I am sure that the full ECPS–monad embedding is not novel, but I connected the dots independently.)

6 CONCLUSION

No single paper could fully explain either delimited continuations or monads, much less the two together. The goal of this report is simply to establish a foundation for further understanding.

For more reading on delimited continuations (and continuations in general) look for work that deals with control flow. As previously mentioned, continuations were first used as meta-theoretic tools, and Strachey and Wadsworth were among the earliest to use continuations [15]. Their goal was to give a semantics for `GOTO` statements. Modern work on delimited continuations is extremely diverse, ranging from automatic differentiation [17] to logic programming [13]. Hands-on exploration of shift and reset can be done in languages like Racket and Scheme.

Further reading on monads can be found throughout the functional programming literature. Languages like Haskell—which uses monads to isolate all effectful code—rely on monads in even the simplest programs, and all research on those languages implicitly involves monads. More theoretical papers using monads often use them to embed effects in languages [11] or to define other more complex abstractions [20].

Finally, some modern ideas build on both delimited continuations and monads, most notably *algebraic effects* [1]. Algebraic effects define *effect operators* and *handlers*, which build on many of the ideas behind shift and reset. These abstractions also isloate programmatic effects in a style very similar to that of monads, and many think they can be replacement for monadic effects in some programming languages.

Delimited continuations and monads are both amazing ideas, and this report only scratches the surface. Still, I hope it is a helpful resource for those who want to learn more about the field of programming languages and need an introduction to some of its ingrained abstractions.

REFERENCES

[1] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, 84(1):108–123, 2015.

[2] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160, 1990.

[3] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190, 1988.

[4] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457, 1994.

[5] Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, 1984.

[6] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[7] James Iry. A brief, incomplete, and mostly wrong history of programming languages, May 2009. URL http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html.

[8] Daan Leijen and Erik Meijer. *Parsec: Direct style monadic parser combinators for the real world*, 2001.

[9] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[10] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[11] Jennifer Paykin and Steve Zdancewic. The linearity monad. *ACM SIGPLAN Notices*, 52(10):117–132, 2017.

[12] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740, 1972.

[13] Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. Delimited continuations for prolog. *Theory and practice of logic programming*, 13(4):533–546, 2013.

[14] Jacob T Schwartz, Robert BK Dewar, Edward Dubinsky, and Edith Schonberg. *Programming with sets: An introduction to SETL*. Springer Science & Business Media, 2012.

[15] Christopher Strachey and Christopher P Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-order and symbolic computation*, 13(1):135–152, 2000.

[16] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.

[17] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3 (ICFP):1–31, 2019.

[18] Jamie Willis, Nicolas Wu, and Matthew Pickering. Staged selective parser combinators. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020.

[19] Li-yao Xia. The reasonable effectiveness of the continuation monad, Oct 2019. URL https://blog.poisson.chat/posts/2019-10-26-reasonable-continuations.html.

[20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.