Side Effects

Amr Sabry, Indiana University

Introduction: Functions and Computational Effects

This section covers basic topics and illustrates the technical notion of a computational effect with several mathematical and programming examples.

In everyday language, a "side effect" refers to the unintended or unforeseen consequences of an action. By analogy, in order to identify the side effect of a given computation, it is first necessary to isolate the "publicly stated" or "publicly expected" actions of the computation. Once this is done, all other actions performed by the computation are actions that occur as side effects, or more accurately these actions represent *computational effects*.

Functions

In set theory, a function is usually defined as a mapping that relates each element in its domain to an element in its range. For example, the function *square* defined over the domain of integers denotes the following infinite set of pairs:

$$\{\ \ldots,\ (-4,16),\ (-3,9),\ (-2,4),\ (-1,1),\ (0,0),\ (1,1),\ (2,4),\ (3,9),\ (4,16),\ \ldots\ \}$$

There is no notion of this mapping being "calculated" in any sense: it just is. Similarly one "knows" from the mapping that the square of 3 is 9: no time, energy, or any other resource needs to be spent to acquire this knowledge.

The mathematical idealization of computation founded on the λ-*calculus* (1) maintains this understanding of computations as mathematical functions. One can use this mathematical framework to reason about computations and as the basis for realizing computations on actual physical machines. But ultimately, programming practice deviates (sometimes significantly) from the set-theoretic view of functions or the mathematical idealization of the λ-calculus. For one thing, an infinite set cannot be directly stored in a

computer; a function is therefore represented using rules which, given enough time, energy, memory, and other resources, can be interpreted by a computer to generate any particular output on demand. It is these rules that we must now discuss in some detail. For concreteness in this section, we express them using the notation of the Java programming language (2).

As a reasonable starting point, the set-theoretic function *square* can be realized in Java using the following method declaration:

```
int square (int n) {
  return n * n;
}
```

The code fragment declares a method called `square` that takes a value called `n` from the domain of Java integers `int` and returns another Java integer whose value is the square of the input.

In an appropriate environment, a computer with an installed Java virtual machine can be used to calculate the value of `square(3)`. This calculation will proceed in elementary steps that require time, energy, memory, and other computational resources on the computing platform to produce the result `9`.

Despite the mismatches between the idealized set-theoretic function and its Java realization, it is common to consider them equivalent at a certain level of abstraction. This intended level of abstraction is reflected in the first line of the method declaration in Java. This *interface* (or *type* or *signature* or *contract*) line:

```
int square (int n)
```

summarizes the publicly stated, computationally relevant, aspects of the method. In this case, it is announced that the method is a computer realization of a pure mathematical map from one `int` to another. The fact that executing the computer realization of the map on a particular input might consume 10% of the battery power on your laptop is something that—although potentially critical knowledge in some circumstances—is not even expressible in the notation used for Java interfaces.

Partial Functions (Time)

Computations take time. At some level of abstraction, we can ignore the time taken by a computation. For example, we can pretend that a calculation that takes one or two microseconds is similar to the "instantaneous" lookup in a mathematical mapping. But we can hardly keep pretending this as the time used by the calculation tends towards infinity.

Consider the following Java method:

```
int collatz (int n) {
  if (n == 1) return 0;
  else if (n % 2 == 0) return 1 + collatz (n / 2);
  else return 1 + collatz (3 * n + 1);
}
```

As evident from the use of `collatz` in the body of the method, this method is *recursive*. It takes an input number `n` and:

- if the input number is equal to one, the method immediately stops with the value zero, which represents in this case the total number of recursive calls performed before stopping;
- if the input number is even, the method divides it by two and recursively invokes itself, adding one to the total number of recursive calls;
- if the input number is odd, the method multiplies it by three, adds one, and then recursively invokes itself, after again adding one to the total number of recursive calls.

It is a famous mathematical conjecture, *the Collatz conjecture*, that this calculation always terminates for all positive integers. One can quickly check a few cases:

- invoking `collatz(3)` iterates through 10, 5, 16, 8, 4, 2, 1, and then stops returning the result 7.
- invoking `collatz(17)` iterates through 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, and then stops returning the result 12.

There is however no known proof that this method always stops. Thus, even though the method has the same interface as the method `square` above, one may not be able to interpret the two interfaces in the same way. To account for the possibility that the calculation never stops, the interface:

```
int collatz (int n)
```

should in general be read as saying:

> Given a particular `int` the method *may or may not* return another `int` in a finite amount of time.

Another way to look at the situation is that we have introduced "time" as an implicit parameter of computations. Some computations as the method `square` update—as a side effect—this time parameter by consuming a few ticks. By general agreement, this consumption is deemed negligible and not "observable"

and does not generally constitute a computational effect. But when the consumption of time is unbounded, it arguably becomes an observable computational effect.

Assignments (Memory)

Computations require memory. Again, at some level of abstraction we can pretend that memory does not exist. But what happens if a computation starts changing external memory locations?

Consider the following variant of the method `square`:

```
int square (int n) {
  count++;
  return n * n;
}
```

As before, the method takes an integer `n` from the domain of Java integers `int` and returns another integer whose value is the square of the input. But, as a side effect the method reads and updates an external reference `count`, which can be used to find out how many times the method was called. Thus if during the execution of a particular program the method `square` is called ten times, then the value of `count` will be ten.

Thus the interface:

```
int square (int n)
```

should now be read as saying:

> Given a particular `int` the method *may possibly* read and update external memory references and then it *may or may not* return another `int` in a finite amount of time.

Another way to look at the situation is that we have introduced "memory" as an implicit parameter of computations. Some computations as the original method `square` might—as a side effect—allocate, update, and read several memory locations (registers, cache entries, stack frames, etc.). By general agreement, these actions on memory are not considered observable and do not generally constitute a computational effect. But when a computation updates an external memory location that other program fragments can inspect, the update of the memory location becomes an observable computational effect.

Runtime Exceptions (Context)

Computations occur in a context. When a computation occurs in the context of a larger computation, it is typically expected to return its result to that outer computation. But what if a computation cannot return

a sensible value to its context and decides instead to abort it?

Consider again the method `square` whose interface is:

```
int square (int n)
```

We initially argued that it is possible to interpret this interface to say that given a particular `int` the method's execution promises to return another `int`. This is clearly an idealized view: in real life, any of an unlimited number of failures may occur in the hardware or underlying software components including the operating system or the Java virtual machine. The Java language collectively refers to these conditions as `Error`s or `RuntimeException`s.

So to be more accurate, and putting together our observations so far, one should interpret the interface:

```
int square (int n)
```

as saying:

> Given a particular `int` the method *may possibly* read and update external references, and then it *may or may not* return another `int` in a finite time or it *may fail to return to its context* due to any of a number of errors and runtime exceptions.

Another way to look at the situation is that we have introduced the "context" as an implicit parameter of computations. It is expected that every inner computation would resume its context with its result. But when an inner computation encounters a fatal error condition, it can ignore the context parameter, which causes the error condition to immediately terminate the entire computation. This escape clearly constitutes an observable computational effect.

Other Computational Effects

The unifying theme in all the above effects is that computations do not happen in isolation in some abstract mathematical world: rather computations are realized in some physical computing platform. The physical interactions of a computation with the computing platform and with other computations may be ignored in some idealized situation corresponding to simple computations, but they become crucial if one is interested in modeling the actual behavior of a sophisticated realistic computation. If these interactions happen "behind the scene" they are considered side effects of the computation.

Without further constraints on computations and their contexts, it is impossible to enumerate all the possible side effects that a computation may have. The reader may find it instructive to consider the following list of effects that could be performed by a method with the interface:

```
int square (int n)
```

Given a particular `int`, the method's behavior may include various combinations of the following actions:

- Reading and updating external memory locations, environment parameters, local or remote file systems or databases, etc.
- Communicating with a user or another local or remote process.
- Suspending itself for a certain period of time or until a certain outside condition happens.
- Resuming other suspended computations.
- Loading, compiling, and executing some other computation.
- Using reflection to examine and perhaps modify its own machine representation.
- Producing multiple answers by arranging for one `int` to be returned now and subsequent answers to be returned when the method is called again (with the same parameters).
- Checkpointing its entire state to a log, which can be examined and resumed later.
- Failing due to any of a number of errors and runtime exceptions.
- Consuming half of the battery life of a laptop and generating enough heat to keep you warm on a cold winter night.

And yes by the way, the method may actually return another `int` in a finite amount of time.

It is clear that the method's interface is nowhere close to giving us a reasonable approximation of the possible behaviors of the method: it sweeps too many things under the rug. Pragmatically, if a programmer invokes a method with the interface:

```
int square (int n)
```

the implicit computational effects of the method might come as a surprise.

The situation is not unique to Java. Almost every other programming language allows similar computational effects to occur implicitly. Even hardware circuits and assembly language instructions often perform implicit effects. For example, an assembly instruction might change the contents of the flag register as a side effect. In fact natural languages also include various "non-compositional phenomena," which are essentially computational effects (3). For this reason, programmers are universally taught to avoid side effects if possible and to be careful when analyzing code that might contain side effects. But when side effects are unavoidable, the solution is to *expose them* as the rest of the article explains.

Type and Effect Systems

A computation that performs effects is arguably harder to understand, analyze, and optimize than a pure computation. But computational effects are necessary in many situations. In such cases, it is of great benefit to expose the relevant implicit parameters (*e.g.*, memory, context, etc.) and describe the computational effects as explicit actions on these parameters. Indeed just as the main uses of a drug and its possible side effects must be publicly declared on its label, the main expected uses of a program and its possible side effects should be declared in its interface for the benefit of both program analysis tools and programmers.

There are two main technologies for exposing effects: *type and effect systems* (4, 5, 6) and *monads* (7). The two approaches are formally equivalent in the sense that it is always possible to translate one presentation to the other (8) but we follow tradition and discuss each separately. We begin with type and effect systems and defer the discussion of monads until the next section. (Uniqueness types (9) and side effect witnesses (10) are two other interesting but less common technologies that are not covered in this article.)

The basic idea of type and effect systems is simply to extend the language used to define interfaces to include additional annotations. In a hypothetical extension of Java we might for example change the interface of `square` as follows:

```
int square (int n) readsAndWritesExternalLocations {
  count++;
  return n * n;
}
```

The additional annotations immediately communicate the possible effects one might encounter by calling the method.

To be reliable these annotations should be *sound* and *precise*:

- **Soundness**: if an effect *may* happen during the execution of the method, it *must* appear in the list of annotations attached to the interface. This is for the same reason that if a drug may have a side effect it should appear on its label.

- **Precision**: the annotations should be as specific and as informative as possible. For example, the annotations for `square` would be more informative if they referred to the particular locations that might be accessed.

Soundness should be a property of every type and effect system. Precision is an engineering trade-off among various constraints and depends on the particulars of each system.

Checked Exceptions in Java. One of the simplest and most popular type and effect systems is the one present in Java, and which tracks the possible exceptions that a computation may raise. We illustrate the ideas with a small but complete example.

Consider the following class that models a simple bank account.

```
class BankAccount {
    private double balance;
    public void deposit  (double amount) { balance += amount; }
    public void withdraw (double amount) { balance -= amount; }
}
```

Each instance of the class corresponds to a particular bank account and each such account maintains a current balance. The methods `deposit` and `withdraw` add and subtract the given amount to and from the balance.

No financial institution is likely to provide a method `withdraw` with the behavior above, which allows the customer to withdraw unlimited funds exceeding the current balance. Some financial institutions may forbid such withdrawals completely; some financial institutions may allow, for their preferred customers, a limited number of such withdrawals as long as they do not exceed a preset maximum; other institutions may automatically check if the customer has another account and withdraw the amount from that other account, etc.

The possibilities are unlimited and it would seem that the method `withdraw` has no sensible action to perform that would be consistent with all the desired possibilities. The way out is for the method `withdraw` to throw an exception indicating some unusual circumstance. It is critical that this exception appears in the interface of the method so that everybody calling the method would be forced to somehow deal with the exception. We can achieve this in two steps. First we declare a new special exception to communicate the nature of the failure:

```
class InsufficientFundsException extends Exception {}
```

Then we modify the method `withdraw` in the class `BankAccount` to use the exception when the requested amount is larger than the available balance:

```
class BankAccount {
    private double balance;
    public void deposit (double amount) { balance += amount; }
    public void withdraw (double amount) throws InsufficientFundsException {
```

```
        if (amount > balance) throw new InsufficientFundsException();
        else balance -= amount;
    }
}
```

The interface of the method `withdraw` now explicitly states that the method may fail by throwing the exception `InsufficientFundsException`, and attempting to compile the method without the `throws` clause will not succeed. In addition the Java compiler will force any other method that calls `withdraw` to deal with the possibility of failure. For example, consider the method `automaticBillPay` below:

```
class Customer {
    private BankAccount checking;
    private BankAccount savings;

    public Customer (BankAccount checking, BankAccount savings) {
        this.checking = checking;
        this.savings = savings;
    }

    public void automaticBillPay (double amount) throws InsufficientFundsException {
        try {
            checking.withdraw(amount);
        }
        catch (InsufficientFundsException e) {
            savings.withdraw(amount);
        }
    }
}
```

The method first attempts to withdraw the amount from the checking account. If that succeeds, then everything is fine. If however the checking account has insufficient funds, the method attempts to withdraw the amount from the savings account. If that latter action fails, then the `automaticBillPay` method itself must fail and hence it must include the clause `throws InsufficientFundsException` in its interface.

Imagine the customer has a rich uncle from whom an unlimited amount of money can be borrowed:

```
class RichUncle {
    public void borrow (double amount) {}
}
```

Then the customer can handle the exceptions as follows:

```
class Customer {
    private BankAccount checking;
    private RichUncle richUncle;

    public Customer (BankAccount checking, RichUncle richUncle) {
        this.checking = checking;
        this.richUncle = richUncle;
    }

    public void automaticBillPay (double amount) {
        try {
            checking.withdraw(amount);
        }
        catch (InsufficientFundsException e) {
            richUncle.borrow(amount);
        }
    }
}
```

In this case, the failure of the method `withdraw` is completely handled by the customer and the method `automaticBillPay` can never fail due to insufficient funds.

In summary, adding the `throws` clause to the interface changes our view of the exceptions that might occur during the execution of the method. Instead of being unexpected effects, these exceptions are now documented in the public interface and their handling is checked and enforced by the compiler. The price one has to pay for exposing the exceptions in the interface is that the composition of computations is now more complicated: not only should the types of the arguments and results match but also every possible computational effect performed by the first method must be appropriately handled by the second method. Java relies on the programmer to manage this complexity but it is possible, in principle, for the compiler to perform this analysis in the background (11).

Other Type and Effect Systems. The original method `withdraw` has another clear side effect:

```
public void withdraw (double amount) {
    balance -= amount;
}
```

It reads and updates the balance! A strong argument could be made that this effect should be exposed in the interface of the method. Unfortunately this is not directly expressible in Java. There are however several type and effects systems designed for exposing and reasoning about such memory accesses (12, 13, 14, 15). Each system targets a particular programming language and is designed with a particular goal that influences the balance of complexity, convenience, and precision. Type and effect systems have also been used to analyze security protocols (16), to reason about concurrency (17, 18), and to model and reason about many other effects.

Monads

This section covers advanced topics and involves issues related to the formal semantics of programming languages and the programming language Haskell (19).

Monads are a mathematical construction (20) that can be used to separate a language into two sublanguages: a pure language with no effects and a computational language in which effects may occur (7). The idea is related to the distinction in most programming languages between *commands* and *expressions* where commands are evaluated for their effects and expressions for their values. What distinguishes monads is that the separation of the two sublanguages is strict and enforced by the compiler. This way, not only do effects become explicit but they are also isolated in a separate sublanguage, which makes it easy to track them and reason about them. Furthermore and quite significantly, monads uniformly model a large collection of effects, which makes both programming and reasoning about programs more modular.

Although the monad construction can be realized in any programming language, it is most effective and clear when presented in the context of a functional language (21). For this reason we present the idea of monads in the programming language Haskell, which has special built-in syntactic constructions to support monadic programming. For convenience, we also use some of the facilities provided by the Glasgow Haskell Compiler (GHC) (22). We note that the use of monads is gaining popularity as an elegant way to express computations with effects. Microsoft Research is developing a programming language F# that includes a construct closely related to monads, "computation expressions." As the MSR researchers describe them, these computation expressions can be used to "express data queries and client/server modalities in AJAX-

style web programming. They enable programmers to write succinct and robust reactive agents through the use of asynchronous workflows." (23)

Haskell: the pure sublanguage of values

A Java method with interface:

```
int square (int n)
```

corresponds to a Haskell function of type `Int -> Int`. Unlike the corresponding Java method, the Haskell function is guaranteed to be almost free of effects. Sure, it may fail due to some runtime error or it may not return an `Int` in a finite amount of time but it is guaranteed to behave the same for every invocation with the same argument. This means that the function cannot read external updatable locations, cannot read files, cannot communicate with a user or another process, cannot use a random number generator, etc. Such are the guarantees of the type `Int -> Int` in Haskell.

Mathematically speaking, Haskell attempts to remain quite close to the idealized model of computation based on the λ-calculus. To a large extent, it is possible to think of a Haskell type as representing a mathematical set. For example, the type `Bool` corresponds to the set containing the two elements `False` and `True`; the type `Int` corresponds to the set of integers; etc. Given two types `a` and `b`, the type `(a,b)` is the type of ordered pairs whose elements are of the respective types; the type `a -> b` is the type of functions mapping elements of `a` to elements of `b`; and the type `[a]` is the type of sequences (lists) whose elements are of type `a`. Many elegant and efficient programs can be expressed in this pure sublanguage of Haskell without ever needing the forbidden effects.

Haskell: the computational sublanguage of effects

A realistic Haskell program needs to perform input/output, access files, and communicate with users and other processes, etc. All these actions are forbidden in the pure sublanguage of values but they can be expressed using individual monads embedded in the computational sublanguage of effects.

State Monad. Let's take another look at the method `withdraw` which, as a side effect, updates the balance:

```
public void withdraw (double amount) {
    balance -= amount;
}
```

Although effects are forbidden in the pure sublanguage of Haskell, it is possible to simulate this computation in the pure sublanguage as follows:

```
withdraw :: (Double,Double) -> ((),Double)
withdraw (amount,balance) = ((), balance - amount)
```

Instead of silently performing an effect to update the balance, the Haskell function takes the old balance as an additional argument and returns the new balance as an additional result, which exposes the implicit memory used to perform the computational effect. This technique is actually the standard way to explain the semantics of assignments using mathematical (denotational) semantics (24), and is generally known as "store-passing style" or "state-passing style."

This simulation of the effect suggests that there is really nothing foundational that forces computations to have effects. And indeed this is the point of the λ-calculus as a universal model of computation. But there are pragmatic reasons to favor the ability to perform updates to external references instead of the "store-passing" simulation. First the simulation is awkward and quickly becomes tedious and unmanageable. Second it is arguably more efficient— at least in some cases— to update a single location in a large external collection than to copy parts of the collection to simulate the update.

These pragmatic considerations led to the introduction of monads in Haskell. Using the state monad the function `withdraw` could be rewritten as follows:

```
withdraw :: Double -> State Double ()
withdraw amount =
  do currentBalance <- get
     put (currentBalance - amount)
     return ()
```

The first line specifies that this function takes a value of type `Double` (the amount to withdraw) and returns a *monadic computation* of type `State Double ()`. This latter type can be read as follows: when this computation is executed, it may read from or write to an external state of type `Double` and then return a value of type `()`. As the actual code shows, the computation first reads the current state to get the current balance, updates the state with the new balance, and then returns `()` to indicate that it is done.

Although this new function looks rather different from the simulation that passes the state around, it is essentially equivalent to it. In this case, the monadic construction is simply a convenient notation to disguise the tedious aspects of passing the store around. But because all manipulation of the state is confined to the monadic sublanguage, it is possible to change the efficiency of this manipulation by changing the monad.

Indeed, although non-trivial to prove (25), it is correct to use an efficient implementation of the monad that updates the location in place:

```
withdraw :: Double -> STRef s Double -> ST s ()
withdraw amount balanceRef =
    modifySTRef balanceRef (subtract amount)
```

In this implementation, `withdraw` takes a `Double` representing the amount to withdraw and a reference to a memory location holding a `Double` representing the current balance. The type of this reference is `STRef s Double` where the `s` parameter is used to track all the uses of this particular reference by the type system. The function returns a monadic computation of type `ST s ()`, which can be read as follows: when executed the computation uses the portion of memory with references indexed by `s` and then returns the value `()`.

Effect Masking and Monadic Encapsulation of State.

In a way that is similar to type and effect systems, the use of the monads exposes the effects of a computation in its type. In both cases, exposing the effects is arguably desirable from a software engineering perspective. But one of the pleasant "side effects" of exposing this information, is that it can be reasoned about and controlled in ways that were not possible had the information remained implicit. In particular, using either type and effect systems or monads, it is possible for the compiler to approximately track the lifetime of every memory reference. This information can be used to assert that a computation neither imports nor exports any memory reference from its context, and hence that this computation can be considered free of effects (26, 12, 14).

In the case of monadic state, this property is assured by tracking the special index `s` mentioned in the type of references and monadic computations as explained above. For example, consider a computation that should remove all duplicates from a large collection of strings. A conventional and efficient implementation of such a computation would use a hash table to mark every string that has already been seen. In other words, the duplicates can be removed in one single pass over the collection assuming the computation has access to an external memory that it can read and update. We do not give the actual implementation but the type of such a computation might be:

```
removeDuplicates :: Hashtable s String -> [String] -> ST s [String]
```

The first argument is the hash table holding the previously seen strings; as its type indicates this hash table occupies a portion of memory indexed by `s`. The second argument is the collection of strings not yet seen.

The result is a monadic computation, which when performed accesses the state indexed by `s` and returns the collection of strings with no duplicates.

Clearly, once the computation is done, all references to the hash table are no longer needed, and the entire memory indexed by `s` can be reclaimed and reused for other purposes. This informal reasoning can be expressed using the special construct `runST` as follows:

```
nodups :: [String] -> [String]
nodups ss = runST (do ht <- newHashtable []
                      removeDuplicates ht ss)
```

The argument to `runST` is the monadic computation that uses the hash table. The fact that the special index `s` occurs neither in the type of the input to `nodups` nor in the type of its output guarantees that the effects of the computations occur strictly internally to the `runST` argument (27). As far as the rest of the world is concerned, the function `nodups` is a pure function, which takes a collection of strings and returns another collection of string performing no effects at all.

Defining Custom Monads. Most Haskell implementations come equipped with a standard collection of built-in monads that can be used to express common effects such as input/output, exceptions, concurrency, etc. In addition, a programmer may define a new notion of effect and use it to extend the class of monads. We illustrate this idea in detail by defining a monad for expressing the effect of *non-determinism* or *backtracking*.

It is often useful to consider situations in which one computation may abstract several choices such that each choice yields one value. This allows a certain style of programming in which several possible solutions are generated and then filtered by additional constraints. For example, consider the following Haskell computation in the monad `Choice` to be defined below:

```
solve :: Choice (Int,Int,Int)
solve = do x <- choose [1..15]
           y <- choose [1..15]
           z <- choose [1..15]
           guard (x + y + z == 15)
           guard (x /= z)
           guard (y == z)
           guard (x * y > 15)
           guard (y * z > 30)
           return (x,y,z)
```

The first line states that the computation returns a triple of integer values after exploring some choices. The actual computation starts by giving possible values for `x`, `y`, and `z`—each ranging from `1` to `15`. The computation then adds several constraints on the desired solution: the sum of the numbers must be equal to `15`, the value of `x` should be different from `z`, and so on. The computation returns the three selected values as its answer. Depending on the particular constraints used to filter the choices, the computation may actually return no solutions or more than one solution. Our particular example was constructed to have the unique solution `(3,6,6)`.

It is certainly possible to write this computation without using monads (28). This is again not a surprise since we have already mentioned that the λ-calculus is a universal model of computation. But using the monad makes this computation quite elegant; in addition, because the manipulation of choices is confined to the monad there are several possible ways to implement the management of choices. We present a very simple implementation below: other more efficient implementations are possible (29, 30):

```
data Choice a = Choices [a]


fail :: Choice a
fail = Choices []


succeed :: Choice ()
succeed = Choices [()]


choose :: [a] -> Choice a
choose vs = Choices vs


guard :: Bool -> Choice ()
guard False = fail
guard True = succeed
```

The first line introduces a new type called `Choice` that represents the new kind of computation we wish to perform. In our simple implementation, computations of this type are internally represented using sequences of values. A computation that fails produces no values, *i.e.*, an empty sequence. A computation that succeeds produces a sequence with just one value `()` indicating success. A computation abstracting several possible values is represented by this sequence of values. The function `guard` takes the result of a comparison and fails or succeeds depending on whether the comparison is `False` or `True`.

So far we have not used the monadic infrastructure at all. In order to embed this new notion of computation in the monadic sublanguage, we must specify the way `Choice`-computations should be composed together in the monadic sublanguage. To this end, each instance of the monad construction must define the functions `return` and `>>=` and guarantee that the definitions agree with the minimal requirements of "composition" explained below:

```
instance Monad Choice where
  return v = Choices [v]
  (Choices vs) >>= f = Choices (concat (map (getChoices . f) vs))
     where getChoices (Choices vs) = vs
```

In more detail:

- the function `return` represents the unit of composition; it explains how to inject a pure value into the world of monadic computations. In our case a regular pure value can be considered as a computation with exactly one choice, *i.e.*, a sequence with one element.
- the function `>>=` explains how to compose two monadic computations. In our case, if we have two computations and the first computation represents several choices, then we should feed each of these choices to the second computation. All the choices resulting from executing the second computation are concatenated together to produce the final list of choices.

This completes the implementation, but one important point remains. For `Choice` to truly deserve to be called a monad, the implementations of `return` and `>>=` must satisfy the following three monad laws:

```
Left identity  :     return a >>= f    =    f a
Right identity :       m >>= return    =    m
Associativity  :     (m >>= f) >>= g   =    m >>= (\x -> f x >>= g)
```

These laws are common to a large class of effects (7). A program or a compiler is always free to use such laws in reasoning about, compiling, or optimizing the monadic sublanguage.

Advanced Applications and Open Problems

Continuations: Effects Expressing Other Effects

Since the introduction of monads in Haskell to express computational effects, an amazing variety of programming constructs were discovered to fit the monadic construction. This naturally raises the question

of whether other programming languages are expressive enough to model such computational effects. The general question of which programming constructs can express others is a well-studied one (31). It has led to results showing that certain computational effects cannot be expressed by others (32).

Quite remarkably, it has been shown that one particular computational effect: *delimited control* (33, 34), can express the same class of computational effects as monads (35). To understand delimited control operations, consider the following example.

$$
\begin{array}{lllll}
 & \langle 5 * (4 * (3 * 2)) & \mid \Box\rangle & & \\
\rightarrow & \langle 4 * (3 * 2) & \mid 5 * \Box, \Box\rangle & (\mathit{Push}) & \\
\rightarrow & \langle 3 * 2 & \mid 4 * \Box, 5 * \Box, \Box\rangle & & \\
\rightarrow & \langle 6 & \mid 4 * \Box, 5 * \Box, \Box\rangle & & \\
\rightarrow & \langle 4 * 6 & \mid 5 * \Box, \Box\rangle & & \\
\rightarrow & \langle 24 & \mid 5 * \Box, \Box\rangle & & \\
\rightarrow & \langle 5 * 24 & \mid \Box\rangle & (\mathit{Pop}) & \\
\rightarrow & \langle 120 & \mid \Box\rangle & &
\end{array}
$$

The example traces a simple calculation using an auxiliary stack. In the line marked *Push*, the context $(5 * \Box)$ is pushed on the stack so that the calculation can focus on the subterm $(4 * (3 * 2))$; when the result of this subterm is available, the context $(5 * \Box)$ is popped from the stack in the line marked *Pop*.

Delimited control is expressed using two operators: one that marks the stack # and one that captures the current stack up to the closest mark (*capture*). For example:

$$1 + \#\ (2 + (\mathit{capture}\ k.3 + (k\ (k\ 5))))$$

whose evaluation proceeds as follows:

$$
\begin{array}{lll}
 & \langle 1 + \# \ (2 + (\mathit{capture}\ k.3 + (k\ (k\ 5)))) \mid \Box \rangle & \\
\rightarrow & \langle \# \ (2 + (\mathit{capture}\ k.3 + (k\ (k\ 5)))) \mid 1 + \Box, \Box \rangle & \\
\rightarrow & \langle 2 + (\mathit{capture}\ k.3 + (k\ (k\ 5))) \mid \#, 1 + \Box, \Box \rangle & \\
\rightarrow & \langle \mathit{capture}\ k.3 + (k\ (k\ 5)) \mid 2 + \Box, \#, 1 + \Box, \Box \rangle & (\mathit{Capture}) \\
\rightarrow & \langle 3 + (k\ (k\ 5)) \mid 1 + \Box, \Box \rangle & \mathit{where}\ k = 2 + \Box \\
\rightarrow & \langle k\ (k\ 5) \mid 3 + \Box, 1 + \Box, \Box \rangle & \\
\rightarrow & \langle k\ 5 \mid k\ \Box, 3 + \Box, 1 + \Box, \Box \rangle & (\mathit{Use}) \\
\rightarrow & \langle 2 + 5 \mid k\ \Box, 3 + \Box, 1 + \Box, \Box \rangle & \\
\rightarrow & \langle 7 \mid k\ \Box, 3 + \Box, 1 + \Box, \Box \rangle & \\
\rightarrow & \langle k\ 7 \mid 3 + \Box, 1 + \Box, \Box \rangle & (\mathit{Use}) \\
\rightarrow & \langle 2 + 7 \mid 3 + \Box, 1 + \Box, \Box \rangle & \\
\rightarrow & \langle 9 \mid 3 + \Box, 1 + \Box, \Box \rangle & \\
\rightarrow & \langle 3 + 9 \mid 1 + \Box, \Box \rangle & \\
\rightarrow & \langle 12 \mid 1 + \Box, \Box \rangle & \\
\rightarrow & \langle 1 + 12 \mid \Box \rangle & \\
\rightarrow & \langle 13 \mid \Box \rangle &
\end{array}
$$

Evaluation proceeds as usual until the line labeled *Capture*. At this point, the portion of the control stack up to the mark is captured and bound to the variable k: this portion of the stack is now under complete program control. The program may use k more than once as in the rest of the example; or it may not use k at all, effectively performing a non-local return; or it may save a copy of k for later use, effectively suspending that computation and resuming it later. In the particular example above, evaluation proceeds as usual until the lines labeled *Use* where the saved portion of the stack is invoked.

As mentioned above, delimited control operators can encode a large number of other computational effects. Although generally complex, it is possible to get an intuitive understanding of this encoding in some special cases, such as the simulation of assignments. The idea in this case is to associate each assignable variable with a distinguished mark on the stack. The current value of the variable is included in the stack frame immediately below the mark. To access the assignable variable (read its current value or update it), the portion of the control stack up to the mark is captured, which exposes the current value of the variable. This value can be manipulated as desired and then the captured control stack is reinstated to resume the computation.

Interaction of Effects

Taking several drugs at the same time is always risky: the interactions among all the side effects are often complicated and poorly understood. Not surprisingly the same situation occurs with computational effects. Combining several computations each performing different effects is often complicated and poorly understood.

When combining two effects, the simplest situation that one might hope for is that the two effects do not interact at all. This would be the case for example if each effect consists of reading and updating distinct memory regions. When effects combine in this way, program fragments that use the individual effects can be combined elegantly with minimal changes.

If however, as is common, the two effects interfere in some way, then a decision must be made about how to combine them. Consider the interaction of a `Choice` computation (as introduced in the previous section) with another computation that reads and updates a memory region. The combined computation performs two computational effects that can be composed in at least two natural ways:

- First choose a computation to perform and then take the memory region as an explicit parameter. In this case all updates performed in this branch remain local to the branch.
- Alternatively, it is possible to first take the memory region as an explicit parameter and then choose a computation to perform; if that computation updates the memory region and then fails, the next computation that is chosen will see the update to the memory region.

Formally each of these choices could be expressed by defining a *monad transformer* for one of the effects and applying it to the other effect. Again this allows program fragments using the different effects to be combined in modular and elegant ways with minimal changes to either fragment (36).

Unfortunately the general situation is more complicated. In some cases, neither monad transformer can be used to achieve the desired behavior. Intuitively this occurs in situations where it is desirable to perform a "portion" of the first effect, then the second effect, and then resume the execution of the first effect. As an example, consider a computation that might raise an exception that should be caught by the dynamically closest handler and that might migrate itself to a different context when it encounters a special mark on the stack (37). Migrating the entire computation first would cause an exception to be caught by handlers in the new context, which is undesirable. Executing the entire computation in the original context is also incorrect. Instead the computation should be executed in the original context as follows: if it raises any exceptions before the special mark on the stack is encountered, the exceptions are handled in the original context. As soon as the special mark is encountered, the exception effect is "suspended" until the computation migrates to the new context and then is resumed in the new context.

In general it is acknowledged that the interactions among effects are poorly understood. For example, Plotkin and Power argue that we currently lack "a precise mathematical basis on which to compare and contrast the various effects." (38). Furthermore, even the precise notion of what is exactly an effect is not well-understood. Indeed although monads were introduced as a construction that models most of the common computational effects, many computational effects are not monads and several new constructions were proposed to model these different classes of effects, *e.g.*, idioms (39) and arrows (40). Such a situation naturally raises the question of what precisely is an effect.

References

[1] H. P. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science (vol. 2): background: computational structures*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[2] Sun Microsystems. Java technology. URL: `http://java.sun.com/`.

[3] C. c. Shan. Linguistic side effects. In Chris Barker and Pauline Jacobson, editors, *Direct Compositionality*, pages 132–163. Oxford University Press, 2007.

[4] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, January 1988.

[5] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In the *IEEE Symposium on Logic in Computer Science*, pages 162–173, June 1992.

[6] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer-Verlag.

[7] E. Moggi. Computational lambda-calculus and monads. In the *IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989. Also appeared as: LFCS Report ECS-LFCS-88-86, University of Edinburgh, 1988.

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

[9] E. de Vries, R. Plasmeijer, and D. M. Abrahamson. Uniqueness Typing Simplified. In Olaf Chitil, editor, *Proceedings Implementation and Application of Functional Languages, 19th International Symposium, IFL 2007, Selected Papers*, LNCS, Freiburg, Germany, September 27-29 2007. Springer.

[10] T. Terauchi and A. Aiken. Witnessing side-effects. *SIGPLAN Not.*, 40(9):105–115, 2005.

[11] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000.

[12] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[13] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[14] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. *SIGPLAN Not.*, 37(5):282–293, 2002.

[15] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2005.

[16] A. D. Gordon and A. Jeffrey. A type and effect analysis of security protocols. In P. Cousot, editor, *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 432–432. Springer, 2001.

[17] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. IC Press, 1999.

[18] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[19] S. L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[20] E. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer Verlag, 1976.

[21] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[22] The GHC Team. The Glasgow Haskell compiler. `http://www.haskell.org/ghc/`.

[23] Microsoft Research. F#. `http://research.microsoft.com/fsharp/about.aspx`.

[24] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1981.

[25] Z. M. Ariola and A. Sabry. Correctness of monadic state: an imperative call-by-need calculus. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 62–74, New York, NY, USA, 1998. ACM.

[26] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp Symb. Comput.*, 8(4):293–341, 1995.

[27] E. Moggi and A. Sabry. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming*, 11(6):591–627, 2001. Cambrige University Press.

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

[29] R. Hinze. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 186–197, New York, NY, USA, 2000. ACM.

[30] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 192–203, New York, NY, USA, 2005. ACM.

[31] M. Felleisen. On the expressive power of programming languages. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 134–151, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[32] J. G. Riecke and H. Thielecke. Typed exeptions and continuations cannot macro-express each other. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 635–644, London, UK, 1999. Springer-Verlag.

[33] M. Felleisen. The theory and practice of first-class prompts. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190, New York, NY, USA, 1988. ACM.

[34] O. Danvy and A. Filinski. Abstracting control. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160, New York, NY, USA, 1990. ACM.

[35] A. Filinski. Representing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457, New York, NY, USA, 1994. ACM.

[36] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, New York, NY, USA, 1995. ACM.

[37] O. Kiselyov, C.-c. Shan, and A. Sabry. Delimited dynamic binding. *SIGPLAN Not.*, 41(9):26–37, 2006.

[38] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 342–356, London, UK, 2002. Springer-Verlag.

[39] C. McBride and R. Paterson. Functional pearl: Applicative programming with effects. To appear in the Journal of Functional Programming.

[40] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.