EECS 662 Homework 3: Non-determinism

J. Garrett Morris

October 23, 2017

Introduction

This homework will help you develop and demonstrate an understanding of *computational effects*, and in particular the interpretation of *non-determinism* in programming languages.

The homework distribution contains three Haskell files:

`Core.hs`	contains the interpreter for the core language (all your changes will go here)
`Main.hs`	contains scaffolding code to run both parsers on either standard input or files
`Sugar.hs`	contains parsing and desugaring functions for an extension of the core language with data types

The `Main.hs` file includes a driver program to simplify interaction with the interpreter. You can build the driver either using the included `Makefile`, or using the following GHC command:

```
ghc -o hw3 --make Main.hs
```

There are two modes of interacting with the interpreter. If you just call the interpreter (without passing any file names on the command line) then it functions as an interactive interpreter. You can write expressions, and those expression will be parsed, checked, and evaluated. For example:

```
$ ./hw3
> 2 + 2
VInt 4
> 2 + True
ERROR: type error
> :q
$
```

Or, you can pass the interpreter a file (or series of files) to interpret. For example, if the file `Add.stlc` contains the text `2+2`, then:

```
$ ./hw3 Add.stlc
VInt 4
$
```

Although it is not visible in these simple examples, the output of the interpreter is actually *a list* of possible results of evaluating the input term. Once you have finished the assignment, you should be able to observe executions such as the following:

```
% ./hw3
> let x = 1 or 2 in let y = 3 or 4 in x + y
VInt 4, VInt 5, VInt 5, VInt 6
>
```

The homework distribution also include the sample source files `Evens.stlc`, `Odds.stlc`, `EvenOne.stlc`, and `Compose.stlc`. The expected result is a comment on the first line of each file.

1 Non-determinism

The key idea in this problem set is that of *non-determinism*. Many programming languages expose non-determinism by giving the programmer access to some external source of randomness—for example, a hardware RNG or software PRNG. However, this does not help us understand programming with non-determinism, it only moves the problem around. We are going to consider a more primitive encoding, in which non-deterministic choice is built into the core language itself. Other languages that follow this approach in include *logic* programming languages, such as Prolog, Datalog, or Curry.

We will add two new terms to our language:

Notation	Meaning
e_1 `or` e_2	Non-deterministically evaluates to either the result of evaluating e_1 or the result of evaluating e_2
`fail`$[t]$	Does not evaluate (t is a type annotation)

The term e_1 `or` e_2 introduces non-determinism to our programming language. For a simple example, the term

```
1 or 2
```

non-deterministically evaluates to *either* `1` or `2`. By itself, this is sufficient to represent many uses of non-determinism in programming languages (in theory, although it would be somewhat impractical). For example, a random coin flip could be encoded by

```
True or False
```

while a random number between 1 and 5 could be described by:

```
1 or 2 or 3 or 4 or 5
```

However, this does not yet by itself present much interest as a programming language feature. To make it more interesting, we will also add a construct `fail`$[t]$, which does not evaluate. In combination with non-determinism, this allows us to express simple search algorithms using the features of our programming languages. The `Even.stlc` example takes this approach to describing a non-deterministic computation of even numbers. Having defined a `even` predicate, as normal, we can then define a function `onlyEven`, which is the identity function for even numbers and fails to reduce otherwise:

```
onlyEven = \x : Int -> if even x then x else fail[Int]
```

Now, for example, the term

```
onlyEven (1 or 2 or 3 or 4 or 5)
```

can evaluate to 2 or 4, but not to 1, 3, or 5.

For this assignment: you will extend the typing and evaluation functions in the provided `Core.hs` to support the `or` and `fail` terms. In the course of extending the evaluation function, you will have to write a new monad instance for the provided `Choices` type.

1.1 Typing

The typing judgment is unchanged from previous assignments (in particular, we are not implementing a type-and-effect system at this point).

Judgment	Meaning
$\Gamma \vdash e : t$	Under typing assumptions Γ, term e has type t

Typing for the non-determinism terms is straightforward. Term e_1 `or` e_2 has the same type as e_1 and e_2 have. (Obviously, they must have the same type, since the term could evaluate to either of them.) The failure term could intuitively have any type (as it does not evaluate); to simplify typing, we attach a type annotation.

$$\frac{}{\Gamma \vdash \texttt{fail}[t] : t} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \texttt{ or } e_2 : t}$$

1.2 Evaluation

We will use the same form of evaluation relation as for the simply-typed λ-calculus:

Judgment	Meaning
$e \Downarrow v$	Expression e evaluates to value v
$[x \mapsto w]e \Downarrow v$	With variable x mapped to value w, expression e evaluates to value v

However, we will rely on the *relational* nature of evaluation more than we have before. In previous systems, evaluation has been a partial function: that is, for a given term e, there is *at most* one value v such that $e \Downarrow v$. Now, we will generalize evaluation to an arbitrary relation: for a given term e, there may be any number of values v such that $e \Downarrow v$.

$$\frac{e_1 \Downarrow v}{e_1 \texttt{ or } e_2 \Downarrow v} \qquad \frac{e_2 \Downarrow v}{e_1 \texttt{ or } e_2 \Downarrow v} \qquad \text{(No rule for } \texttt{fail}[t]\text{)}$$

Because it has two possible evaluations, there are two evaluation rules for $e_1 \texttt{ or } e_2$. Either of these rules can be used in the course of larger derivations; for example, both the following are valid derivations of the evaluation relation:

$$\frac{\dfrac{\dfrac{}{1 \Downarrow 1}}{1 \texttt{ or } 2 \Downarrow 1} \quad \dfrac{}{3 \Downarrow 3}}{(1 \texttt{ or } 2) + 3 \Downarrow 4} \qquad \frac{\dfrac{\dfrac{}{2 \Downarrow 2}}{1 \texttt{ or } 2 \Downarrow 2} \quad \dfrac{}{3 \Downarrow 3}}{(1 \texttt{ or } 2) + 3 \Downarrow 5}$$

The term $\texttt{fail}[t]$ does not evaluate, so the following is the only valid derivation of the evaluation relation for the given term:

$$\frac{\dfrac{\dfrac{}{1 \Downarrow 1}}{1 \texttt{ or } \texttt{fail}[\texttt{Int}] \Downarrow 1} \quad \dfrac{}{3 \Downarrow 3}}{(1 \texttt{ or } \texttt{fail}[\texttt{Int}]) + 3 \Downarrow 4}$$

The remaining typing rules are unchanged; however, in understanding them, you must keep in mind that we are no longer thinking of evaluation as a function. For example, consider the rules for addition and function application.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow w \quad [x \mapsto w]e \Downarrow v}{e_1\ e_2 \Downarrow v}$$

Previously, we would have assumed that there was at most one value v_1 such that $e_1 \Downarrow v_1$. However, there now may be many such values, and each gives rise to a different valid evaluation of $e_1 + e_2$. Similarly, the function term e_1 may have multiple valid evaluations in the application $e_1\ e_2$, and so the application may have many valid results.

1.3 Choice as a Monad

We can imagine several options for implementing non-determinism. For example, we could thread a source of randomness through the interpretation, using it to choose between alternatives at each $e_1 \texttt{ or } e_2$ expression. Or, we could always choose the left-hand argument. However, either of these approaches would make it difficult to implement the behavior of $\texttt{fail}[t]$ properly, as they might require arbitrary backtracking to find random choices that evaluated fully.

An alternative approach is to have evaluation return *the set of possible results* at each point in the computation, rather than picking a single result. Doing so has several advantages. First, it naturally represents failure without having to implement an (explicit) search strategy. Second, it has a natural monad instance, and so makes implementation easy. Finally, although it might seem less efficient, in a lazy language like Haskell it costs no extra performance. (Why not?)

Consider the monad operations a set of values; they will have types like

$$\texttt{return} :: t \texttt{ -> } [t]$$
$$(\texttt{>>=}) :: [t] \texttt{ -> } (t \texttt{ -> } [u]) \texttt{ -> } [u]$$

(We are using lists rather than true sets to simplify the implementation.) In each case, there is only one reasonable definition with the given type: `return` should produce a singleton list, while `xs >>= f` should return the (concatenated) result of applying `f` to *each element of* `xs`.

The homework distribution includes a datatype `Choices t`, which wraps a list of `t` values. It includes the skeleton of a `Monad` instance for `Choices`, as well as the requisite `Functor` and `Applicative` instances.

> ***You may not use Monad instances from the Prelude (i.e., use the >>=operator) in implementing >>=for*** `Choices`**.**

However, you may use other Prelude functions if you find them helpful.

Several notes:

- Because we are using lists to implement non-determinism, the same result may appear multiple times in the result of evaluation. You do not need to filter these duplicates.
- Also, this means your order of results may differ from that given in the test files; so long as all the same results are present, order is not significant.

Submission Instructions

Your submission should include the three Haskell files from the original homework distribution, modified as necessary for your solution, the test files from the original distribution, and (at least) two new test cases, stored in separate files. Each of your test files should being with a comment line (i.e., a line beginning with two dashes) containing their expected result. The provided test follows this format; check there for details. Please submit these as a single file (`.zip` or `.tar`), via Blackboard.

(Provisional) point allocation

Type checking	10
Monad instance	20
Evaluation	20
Test cases (2)	8
Total	58