CS4450 Lecture 27 Notes

Monday, October 22, 2018 3:01 PM

Error Checking & Monads

x ; y
id

x, id = x -- You can say that x followed by id is the same thing as x, and id followed by x is the same thing as x.
id, x = x

-- The semicolon is associative. (x;y);z = x;(y;z)

Skip
x = x

Interpreter so far:

- Our current JavaScript-like interpreter allows us to write recursive functions.
- However, if we declare a function on the fly without it existing, the program will crash when it tries to do an environment lookup.

Errors

- Errors are an **important aspect of computation**.
- They are typically a **pervasive feature of a language**, because they affect the way every expression is evaluated. For example, consider the expression *a + b*
- If *a* or *b* raise errors then we need to deal with that.

How do you report the type of error that it was?

In C, functions are called and they all have a return code (0 is usually ok, non-zero is usually not ok).

Maybe

- The Maybe datatype provides a useful mechanism to deal with errors:
  data Maybe a = Nothing | Just a -- The Nothing is an error, but the Just a is a good result.

Representing Errors

Interpreter that deals with Errors

- Using Checked, we can reimplement the eval function to deal with errors.

eval :: Exp -> Env -> Checked Value

What kind of errors can we have in the current interpreter?

Undefined Variables

- Dealing with undefined variables, we can provide an error message.

Propagating errors

One way of error checking. It works but it's a bit of a mess. Basically, you check after every evaluation/case if it's acceptable and return an error if it's not.

Before Checked, we could evaluate binary operators in one line. After Checked, our code is bloated when we implement it.

Spotting the pattern

- We seem to have something like this:

  case first-part of
    Error msg -> Error msg
    Good v -> next-part v

  How can we capture this pattern as reusable code?

bind :: Checked a -> (a -> Checked b) -> Checked b

We have this pattern, we'll call it **bind**.

Spotting the pattern

- Use a higher-order function!

```
first-part >>= next-part =
  case first-part of
```

Creating auxillary definitions

Rewriting Evaluation

- Here is the new version (4 cases) of evaluation (I didn't type them up, but they're available on the slides on Dr. Harrison's website).

Propagating errors can be done using bind, and the use of it may not be immediately intuitive.

Monads

- Monads are a structure composed of two basic operations (**bind** and **return**).