EECS 662

Programming Languages

Project 3 - Functions and Elaboration

Project 3 – Functions and Elaboration EECS 662 - Programming Languages

The objective of this project is to add dynamically scoped and statically scoped strict functions to BAE and introduce elaboration to the interpretation process. You will first define an interpreter that adds functions, Booleans, and a conditional to BAE while removing the `bind` expression. You will then define an interpreter that uses elaboration to define interpretation of `bind` in terms of function application.

To aid in your quest, the file p3.hs implements Haskell data types and function signatures needed for this project. Look at this file carefully before you start as this project requires two abstract syntaxes. Data types cannot share constructor names, so a tick is used to distinguish constructors. I am not providing a parser for this project because parsers remain evil.

Exercise 1

In this exercise you will write an interpreter for a modified FBAE language presented in our text that does not include the `bind` construct, but does include first-class functions. Following is the grammar for this language that we will call FAE:

```
FAE ::= number | id |
      FAE + FAE  | FAE - FAE |
      lambda id in FAE | FAE FAE |
```

CFAE has numbers, dynamically scoped, first-class functions with strict evaluation semantics and no `bind`. Your interpreter will use deferred substitution for efficiency but will not require closures as it is dynamically scoped. Perform the following:

1. Write a function, `evalDynFAE :: Env -> FAE -> (Maybe FAE)` that evaluates its second argument using the environment provided in its first and returns a `FAE` AST structure.

Exercise 2

In this exercise you will write an interpreter for a modified FAE language from the previous exercise that is statically rather than dynamically scoped. You will need to add closures and values to the interpreter to accomplish this goal.

1. Write a function, `evalStatFAE :: Env' -> FAE -> (Maybe FAEValue)` that interprets its second value using the environment provided in its first. This evaluator needs to return a value rather than a FAE expression to implement static scoping using closures.

Exercise 3

In this exercise you will write a pair of interpreters for a an extension of the FAE language that includes the `bind` construct. This new language will be called FBAE. The trick is that for this exercise you will not write another interpreter at all. Instead you will write an elaborator that will translate FBAE language constructs into CFAE constructs, then call the FAE interpreter. The new language CFBAE has the form:

```
FBAE ::= number | id |
      FBAE + FBAE  | FBAE - FBAE |
      bind id = FBAE in FBAE |
      lambda id in FBAE | FBAE FBAE |
```

1. Write a function, `elabFBAE :: FBAE -> FAE`, that takes a `FBAE` data structure and returns a semantically equivalent `FAE` structure. Specifically, you must translate the `bind` construct from CFBAE into constructs from `FAE`.

2. Write a function, `evalFBAE :: Env' -> FBAE -> (Maybe FAEValue)` that combines your elaborator and statically scoped `FAE` interpreter into a single operation that elaborates and interprets a `FBAE` expression.

The `FBAE` interpreter introduces elaboration to the `FAE`interpreter by using a function that transforms `FBAE` abstract syntax into `FAE` syntax before evaluation. Most of this translation is routine - there are shared constructs in the two languages. For `bind` we have to do a bit more work. Thankfully, not too much more.

As discussed in class, the `bind` construct can be elaborated to an application of a function. Specifically:

```
bind id = t1 in t2 == ((lambda id t1) t0)
```

Thus, to evaluate a `bind` expression in `FBAE`, one need simply translate it into a function application in `FAE` and execute the result.

Exercise 4

Now for something completely different. Let's add Booleans to our language by elaborating Boolean operations to lambdas. We'll extend the language from the previous exercise leaving `bind` as is. The new language has the form:

```
FBAEC ::= number | id |
        FBAEC + FBAEC  | FBAEC - FBAEC |
        true | false
        FBAEC && FBAEC | FBAEC || FBAEC | FBAEC <= FBAEC | ~FBAEC
        if FBAEC FBAEC FBAEC |
        bind id = FBAEC in FBAEC |
        lambda id in FBAEC | FBAEC FBAEC |
```

1. Write a function `elabFBAEC :: FBAEC -> FAE` that translates a `FBAEC` term into an equivalent `FAE` term. Specifically, each list operation must be translarted into an equivalent `lambda` term and each `bind` must be translated as above.
2. Write a function `evalFBAEC :: Env' -> FBAEC -> (Maybe FAEValue)` that combines your elaborator and `FAE` interpreter into a single operation that elaborates and interprets a `FBAEC` term.

We're going to use a technique called Church Booleans to implement Boolean values using functions. Let's start by defining values for `true` and `false`:

```
true = lambda t in lambda f in t
false = lambda t in lambda f in f
```

`true` is a two argument function that returns its first value while `false` returns its second value. Remember that lambdas are values - you cannot interpret these lambda expressions further. They are literally the values for `true` and `false`. Kind of weird, but hang with me.

Let's think about `&&` as an example of how to implement the other problems.

```
&& = lambda x in lambda y in x y false
```

The `&&` function is implemented like a short-circuit conjunction. It takes two arguments and applies the first to the second and `false`. Assume that the first argument is `true`, then `true` applied to `y` and `false` is `y`. If `y` is `true`, then `x && y` is `true`. If `y` is `false`, then the the second argument is returned and the result is always `false`.

`||`, `~`, and `if` can be implemented similarly, but I will leave those to you. If you come up with something more complicated than `&&` for any of these things you are doing something wrong.

Notes

This project looks long. It's really not. Most of the changes aside from the recursive function processing are trivial. Don't be intimidated and just do things step-by-step. Define the first interpreter with dynamic scoping first, then add static scoping for Exercise 2. The elaborator in Exercise 3 need only translate `bind` whlle the elaborator in Exercise 4 adds Booleans. Once you get the definitions, the elaborator is not difficult to build.