Mini Project 4 - Assignment and State

EECS 662 - Programming Languages

Spring 2013

The objective of this miniproject is to add assignment, state, and sequencing the CFWAE language. You will define an interpreter for CFWAE/S, an extension for CFWAE that includes state and sequencing. Your interpreter will differ from the BCFWAE interpreter developed in class in that *every* defined variable – `with` and `fun` – will have an associated location when it is defined. We will use a single `assign` command to change the value referenced by a variable.

Exercise 1 *In this exercise you will write a parser and interpreter for a modified CFWAE language that allows assignment and sequential execution. Table 1 defines the grammar for this language that we will call CFWAE/S.*

CFWAE/S ::=	*number* |
	id |
	{ + *CFWAE/S CFWAE/S* } |
	{ - *CFWAE/S CFWAE/S* } |
	{ ∗ *CFWAE/S CFWAE/S* } |
	{ / *CFWAE/S CFWAE/S* } |
	{ with { *id CFWAE/S* } *CFWAE/S* }
	{ fun *id CFWAE/S* } |
	{ *CFWAE/S CFWAE/S* } |
	{ if0 *CFWAE/S CFWAE/S CFWAE/*
	{ seq *CFWAE/S CFWAE/S* } |
	{ assign *id CFWAE/S* }

Table 1: Grammar for CFWAE/S where *id* is an identifier and *number* is an integer.

CFWAE/S's FEATURES INCLUDE *static scoping, first-class functions and closures, and strict evaluation semantics. Your interpreter will also include assignment (the* `assign` *operation) and sequential execution (the* `seq` *operation).*

Unlike the interpreter we are developing in class, every element of the CFWAE/S environment will contain a location. This simplifies both the environment and how it is used, but eliminates some functionality. Store elements may contain any value including another location. As a result, any time the value of a variable is needed its location is retrieved from the environment and used to retrieve its value from the store.

1. *Define types for representing the abstract syntax, values, environment, and store associated the CFWAE/S language. Your* `CFWAE/S-Value` *type must minimally define, but is not limited to numbers and closures. You may or may not need to include other values. Your your environment type must represent pairs of identifiers and locations while your store type must represent pairs of locations and values.*[1]
2. *Write a function,* `(parse-cfwaes expr)`*, that accepts the concrete syntax of CFWAE/S and generates a CFWAE/S data structure representing it. If your parser fails, it should throw an error using the* `error` *function discussed in class.*
3. *Write a function,* `(interp-cfwaes expr env sto)`*, that takes a CFWAE/S data structure, environment and store. It should interpret the CFWAE/S data structure, and return a value and store as a pair. Using the* `Value`×`Store` *structure from your text is an excellent way to do this. If your interpreter fails, it should throw an error using the* `error` *function discussed in class.*
4. *Write a function,* `(eval-cfwaes expr)` *that combines your parser and interpreter into a single function that parses and then interprets a*

[1] It would be a good idea here to follow the directions indicated in code from the textbook.

CFWAE/S expression. It should return only the resulting value, not the value/store pair generated by the interpreter. The interpreter should be called from within `eval-cfwaes` *with an empty environment and store. Your interpreter should not be called if the parser fails.*

We have implemented the interpretation of all constructs in CFWAE/S except `seq` and `assign`. You should know how to implement strict evaluation, `with`, `if0`, and fist-class functions. However, you will need to modify your interpreter definitions to handle both an expression and store as inputs and return the value/store pair.

THE `seq` EXPRESSION IS relatively simple to implement. The construct:

`{seq` *expr*0 *expr*1`}`

interprets *expr*0 followed by *expr*1 and returns the result of interpreting *expr*1. Note that the store that results from interpreting *expr*0 must be passed to *expr*1 as interpreting *expr*0 may change the values of identifiers. If you don't do this, then the interpretation of one expression can never have any impact on the expression following.

THE `assign` EXPRESSION INVOLVES calculating a value and storing that value in a location. In the term:

`{assign` *x expr*`}`

x must be an identifier while *expr* may be any expression. *expr* is interpreted first returning a value and store. Any identifiers appearing in *expr* are resolved to values by retrieving their location from the environment and value from the store. The resulting valued is stored in the store location associated with *x* in the environment. The identifier, *x*, is treated as a location while identifiers in *expr* are resolved to values. Specifically, *x* is not evaluated, but simply looked up in environment to get a location where the result of evaluating *expr* is stored. The `assign` operation can be defined:

`{assign` *x expr*`}` $\equiv$ `{setbox` *x expr*`}`

using our BCFWAE language from class.

Given how identifiers work, The `with` expression must be updated to handle allocation of store locations in addition to creating new entries in the current environment. In the expression:

`{with {`*x expr*`}` *body*`}`

x is an identifier whose value will be determined by *expr* that may be referenced in *body*. In previous interpreters, *x* would be added to the environment associated with the value determined by interpreting *expr*. This remains true, except that *x* will refer to a new location in the store and the value will be stored in that new location. When the `with` is evaluated, a new entry must be added to the store, associated with the identifier, and initialized with the value found by interpreting *expr*. The `with` expression from CFWAE/S can be defined:

`{with {`*x expr*`} body}` ≡ `{with {`*x* `{newbox` *expr*`}} body}`

using the `with` expression and `newbox` expressions from our BCFWAE language in class.

Remember also that evaluating *expr* may change the store. Thus, the store that results from interpreting *expr* must be the store used when interpreting *body*.

An issue that pervades the entire interpreter for CFWAE/S is the threading of state. Even simple expressions such as binary operations must make sure that updates to the store are propagated. If an expression involves multiple sub-expressions that may be evaluated, the store must be passed from one interpretation to the next as sub-expressions are evaluated. For example, in:

```
{+ {f x} {g y}}
```

interpreting `{f x}` may update the environment in some way. If `f` has the definition:

```
{fun x {assign y x}}
```

then the value of `y` changes during interpretation. This new value must be propagated to the interpretation of `{g y}` using the updated store.

Anytime an expression involves interpretation of multiple sub-expressions, threading state by updating the store will be an issue. We have seen this play out for `with` and `+`, but it will apply to other expressions as well. For the purposes of this project, always assume that sub-expressions are evaluated from left to right. Thus, the store will propagate right from the leftmost sub-expression through to the last sub-expression.

Notes

There is only one exercise in this miniproject.

PAY CLOSE ATTENTION to the `Value*Store` data type and the pattern that emerges when using it. The pattern:

```
(type-case CFWAE/S-Value (interp expr env sto)
  (v*s (the-value the-sto)
    use the-value and the-sto in the code body
    ))
```

binds `the-value` and `the-sto` to the value and store results from `interp`. They can be used like they were defined using a `let` or `local` and will go away with the context closes. This is extremely handy. See class notes as well as the book in Chapter 13 to see how it is used.

THINK CAREFULLY ABOUT how function values are handled and how that changes CFWAE/S. They are stored exactly like other values in the store. Don't think too hard about this.

DO NOT THINK AT ALL about how the store behaves computationally. It is inefficient and we know that. Don't try anything crazy to somehow make it work more like memory.

THIS MINIPROJECT IS LOOSELY BASED ON Shriram Krishnamurthy's homework assignments associated with *Programming Languages: Application and Interpretation*. Any errors in the writeup are exclusively mine.