Mini Project 2 - Adding Functions and Elaboration

EECS 662 - Programming Languages

Spring 2013

The objective of this project is to add dynamically scoped, strict functions to WAE and introduce elaboration to the interpretation process. You will first define an interpreter that adds functions and a conditional construct to WAE while removing the `with` expression. You will then define an interpreter that uses elaboration to define interpretation of `with` in terms of function application and `cond0` in terms of `if0`.

Exercise 1 *In this exercise you will write an interpreter for a modified FAE language presented in our text that does not include the* `with` *construct, but does include first-class functions and an* `if0-then-else` *construct. Table 1 defines the grammar for this language that we will call CFAE.*

CFAE ::=	*number* |
	id |
	{ + *CFAE CFAE* } |
	{ - *CFAE CFAE* } |
	{ * *CFAE CFAE* } |
	{ / *CFAE CFAE* } |
	{ `fun` *id CFAE* } |
	{ *CFAE CFAE* } |
	{ `if0` *CFAE CFAE CFAE* }

Table 1: Grammar for CFAE where *id* is an identifier and *number* is an integer.

CFAE's features include *dynamic scoping, first-class functions and strict evaluation semantics. In addition your interpreter will use deferred substitution for efficiency. The following tasks will step you through the development of the interpreter:*

1. *Define types for representing the abstract syntax and deferred substitutions associated the CFAE language. Your deferred substitution type must represent pairs of identifiers and values.*
2. *Write a function,* `(interp-cfae expr ds)`*, that takes a CFAE data structure and deferred substitution list, and interprets it using dynamic scoping.*[1] `interp-cfae` *should return a value as a CFAE term. If your interpreter fails, it should throw an error using the* `error` *function discussed in class.*

[1] The deferred substitution interpreter from class is dynamically scoped

Modifying the WAE data type from your previous project to handle CFAE expressions is reasonably simple. You will need to remove the `with` expression and add new cases for functions, applications, and `if0`. Note that much, if not all code associated with deferred substitution and values can be taken directly from class notes or from the text.

We have discussed the execution of all constructs in CFAE in class except the `if0` expression. The term:

{ `if0` *c t e* }

is interpreted much like a traditional C or C++ `if` statement, only backwards. Specifically, *c* is evaluated and if it is zero, then *t* is evaluated and returned. Otherwise, *e* is evaluated and returned. For example, the expression:

```
{if0 {- 1 1} 0 {g 3}}
```

evaluates to `0` because `{- 1 1}` is `0`. In contrast, the expression:

```
{if0 {- 1 0} 0 {g 3}}
```

evaluates to `{g 3}` because `{- 1 0}` is `1`.

You might want to think for a minute why we have `if0` and not a regular `if` that checks if its first argument is true or false. What would that do to your AST data structure, concrete syntax, and value type?

CONTINUE TO USE the operator table code from your previous project to implement addition, subtraction, multiplication and division.

THERE IS A NEW TYPE ERROR that you need to handle in CFAE that is not present in your FWAE interpreter. As noted in class, one cannot call a number like a function nor can arithmetic operations be performed on functions. Thus, built-in operations that require arguments to be of a specific type should check argument types at run time. This is as simple as evaluating argument expressions and throwing an error if they are of the wrong type.

Exercise 2 *In this exercise you will write an interpreter for a an extension of the CFAE language that includes the* `with` *construct and a* `cond0` *construct much like the Scheme* `cond`*. This new language will be called CFWAE. The trick is that for this exercise you will not write another interpreter at all. Instead you will write an elaborator that will translate CFWAE language constructs into CFAE constructs, then call the CFAE interpreter.*

1. *Define a type for representing the abstract syntax for CFWAE. You will need to add a* `with` *expression and a* `cond0` *expression to the abstract syntax for the CFAE syntax from the previous problem.*

2. *Write a function,* `(elab-cfwae expr)`*, that takes a CFWAE data structure and returns a semantically equivalent CFAE structure. Specifically, you must translate the* `with` *and the* `cond0` *constructs from CFWAE into constructs from CFAE. The* `with` *expression will translate into a function application while the* `cond0` *expression will translate into nested* `if0` *expressions.*

3. *Write a function,* `(eval-cfwae expr)` *that combines your elaborator and CFAE interpreter into a single operation that elaborates and interprets a CFWAE expression. In addition, your interpreter should have access to a collection of pre-defined symbols – functions and numbers – that will be available to all programs. This collection of pre-defined*

```
CFWAE ::=
  number | id |
  { + CFWAE CFWAE } |
  { - CFWAE CFWAE } |
  { * CFWAE CFWAE } |
  { / CFWAE CFWAE } |
  { fun id CFWAE } |
  { CFWAE CFWAE } |
  { if0 CFWAE CFWAE CFWAE }
  { with { id CFWAE } CFWAE }
  { cond0 { CFWAE CFWAE }* CFWAE }
```

Table 2: Grammar for CFWAE where *id* is an identifier and *number* is an integer.

> *items is typically called a* prelude. *Your prelude should minimally define the constant* `pi` *and two functions,* `area` *and* `inc`*, that define an area function for circles and an increment function respectively.*

THE CFWAE INTERPRETER INTRODUCES three new concepts to the CFAE interpreter – `with`, `cond0`, and a prelude. The first two additions are done by writing a function that transforms CFWAE abstract syntax into CFAE syntax before evaluation. Like all of our interpretation functions it should be structured using a `type-case` that examines CFWAE abstract syntax and transforms it into CFAE syntax. Most of this translation is routine – there are shared constructs in the two languages. For `with` and `cond0`, we have to do a bit more work. Thankfully, not too much more.

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

`{with {id` *expr*0`}` *expr*1`}` ≡ `{{fun id` *expr*1`}` *expr*0`}`

Thus, to evaluate a `with` expression in CFWAE, one need simply translate it into a function application in CFAE and execute the result.

The `cond0` construct is similar. It works much like a Scheme `cond`, selecting the expression associated with the first expression that evaluates to `0`. The most substantial difference is the presence of a default. Thus:

```
{ cond { c0 e0 }
       { c1 e1 }
         ...
       { cn en }
       ed }
```

evaluates to the *ek* associated with the first *ck* that evaluates to `0`. If no *ek* is true, then the default case *ed* is used. You'll need to figure out what the elaboration transformation will be, but it is not difficult if you think about how an `elsif` construct might work.

THE ADDITION OF A PRELUDE makes our interpreter more realistic by adding a collection of pre-defined functions and numbers that are available to all programs. Thankfully, it's not particularly difficult to do using constructs that are defined for deferred substitution already. One need only provide an initial value for the substitution that contains things defined in the prelude. Note however that there is no *ds* argument to either evaluation function. Thus, they are requires to find their prelude definitions elsewhere.

Notes

You can get quite a bit of your code from class notes or the text. Specifically, the definitions for deferred substitution and value structures can be used directly. Furthermore, you should be able to reuse quite a bit of the code from earlier interpreters. The definitions of things like multiplication have not changed since their initial implementation. If you continue to use your binop implementation from the last project, you're going to get quite a bit of code for free. If you want to use direct implementation instead of the binop table, feel free to do so.

The elaboration of `cond` expressions to `if` expressions is actually how most Scheme and Lisp interpreters work. `if` is called a *special form* because it is not a strict function. By implementing `cond` using `if`, language implementers choose the simplest expression to implement as a special form and then build others from it.

To give you a rough idea of the effort required for this miniproject, my code is about 100 lines long and took me rougly an hour to write and debug. I view this as a moderately difficult project at this point in the semester. Do not put it off.

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.