Call-by-Name Loot (an attempt)

Yichi “Rocky” Zhang

Github Repository

Abstract:

In this project, I attempted to implement the compilers and interpreters for a call-by-name (CBN) version of Loot. Concretely, this means the evaluation of each of the bolded expressions in Table 1 was to be computed each time they were needed. This can also be seen as their evaluation being delayed until their values are needed for computation. In most implementations, this involved altering the parser itself to parse the bolded expressions into thunks (i.e. λ-expressions that take no arguments). By the same token, the parser was also usually made to add `App` tokens that would result in an application of a result if the result was guaranteed to be a thunk. For instance, since `cons` was made to delay its arguments by making them thunks, it was guaranteed that `car` would return a thunk. Hence, applying the result of `car` would yield the desired result.

Table 1. Expressions of Interest

“Delayed” Expressions	Expressions that Force Evaluation

`(struct Prim1 ('box e) )`
`(struct Prim2 ('cons e1 e2))`
`(struct Let (x e1 e2) )`
`(struct App (e es) )`
`(struct Match (e ps es) )`
	`(unbox e)`
	`(car e)`
	`(cdr e)`
	`(var x)`

Table 2. Test Results:

Attempt:	Interpreter:	Compiler:
1	9/161	9/99
2	9/161	3/99
3	NA	5/63

The interpreter was straightforward because the implementation language (Racket) was able to handle λ-expressions as well as thunks in a straightforward way, capturing the environments automatically when needed. This was a benefit of using an implementation language with similar features to Loot. As such, the compiler was able to pass all of the relevant tests (see Table 3) that verified call-by-name behavior. The compiler was a disaster for the opposite reason. Ultimately, I was unable to get the compiler working with Iniquity level code, meaning user-defined functions would not compile correctly. I’m still not sure of the exact reason, but the problem stems being unable to sufficiently determine the environment of nested thunks (λ-expressions) at compile time.

Table 3. Tests for CBN Behavior		
Code	CBN Result	Loot Result
`(let ((x (zero? #t))) 42)`	42	Error

`(let ((x (cons 1 (cons 2 '()))) )`
`  (let (( y (cons (zero? #t) x) ))`
`  (car (cdr y))))`
	1	Error

`(define (k x y) x)` `  (k 25 (zero? #t))`	25	error

`(define (inf-seq a0 f) (cons a0 (inf-seq (f a0) f)))`
`(define (inf-seq2 a0 a1 f) (cons a0 (inf-seq2 a1 (f a0 a1) f)))`
`(define (nth seq n)`
`  (if (zero? (sub1 n))`
`  (car seq)`
`  (nth (cdr seq) (sub1 n))))`
`(let ((fib (inf-seq2 0 1 (lambda (x1 x2) (+ x1 x2))) ))`
`  (nth fib 10))`
	34	error

Attempt 1:

1.1 Parser & Interpreter:

My first thought was to alter the parser to add tokens to denote thunks to relevant expressions during parsing before interpreting the thunks as λ-expressions in the interpreter. It turns out that one can avoid adding this new structure by simply "casting" expressions during parsing. Specifically, the parser was adjusted to add tokens (Lam) to create thunks around expressions that would need to have CBN behavior. `App` tokens were added around expressions whose evaluation were being forced. The new changes to the parser are summarized below.

```
[('box e)                (Prim1 'box (parse-thunk e))]
[('unbox e)              (App (Prim1 'unbox (parse-e e)) '())]
[('car e)                (App (Prim1 'car (parse-e e)) '())]
[('cdr e)                (App (Prim1 'cdr (parse-e e)) '())]
[('cons e1 e2)           (Prim2 'cons (parse-thunk e1 env) (parse-thunk e2 env))]
[('let x e1 e2)          (Let x (parse-thunk e1 env) (parse-e e2 (cons (cons x e1) env)))]
[(e es)                  (App (parse-e e env) (map parse-thunk es env))]
[('match (cons e ms))    (parse-match (parse-thunk e env) ms)]

(define (parse-thunk e env) (Lam (gensym 'thunk) '() (parse-e e env)))
```

By parsing langauge features differently by using existing features, we avoid much of the work in the interpreter. I essentially implemented CBN semantics by changing the way the syntax structure was parsed. A small change had to be made in `interp-var` that applied a Racket λ-expression if it could be found in the environment because all expressions bound to variables are thunks until the variable is used.

It should be noted that the tests that the interpreter fail on are all expected to fail. For example, `(let ((x (cons 1 2))) (eq? x x))` does not return true because `(cons 1 2)` is evaluated separately for each evaluation of `x`, so each `x` is in a different memory address, meaning it fails for the same reason `(eq? (cons 1 2) x)(cons 1 2)` fails in Racket. Similar can be said for the `vector-set!` cases. Some cases, like those on line 97 and 181 of the test-runner, fail because the testing library is directly looking at a cons cell, which in our implementation will always be holding a thunk.

1.2 Compiler:

I had a dream that the compiler would be just like the interpreter, hoping I could approach them the same way. Recall, due to the parsing, a variable in the interpreter was just the application of some thunk. Thus, I needed `compile-variable` to call `compile-app`. Unfortunately, `compile-app` takes an expression as an argument, and there was no parameter for the expression corresponding to the variable in `compile-variable`, so I thought I just needed to make the environment richer. In the code snippet above, you might have noticed the `env` variable being passed in each `parse-e` call. Instead of having a compile-time environment that was a list of identifiers (symbols), I aimed to also include their respective expressions as a pair. I thought I could syntactically determine this during parsing, and I would be able to compile any variable so I thought. This certainly helped get me off the ground for `let`.

I think the fundamental reason why this does not work is because thunks are λ-expressions, which are all generated (partially) in assembly without knowing what their arguments will be. The second attempt expands more on this.

Attempt 2:

Discussion of the interpreter will be omitted because there was not much to improve on. At some point, I believed that if all thunks were bound to a variable via a let-expression, then all variables would have to be bound. I thought if all variables were bound, I would be able to syntactically determine the expression corresponding to a variable in a way that I failed to do with a richer environment above. To do this, I compiled `box` and `cons` into `let` as shown in the snippet below.

```
[(box e)        (let ((esym (gensym 'e)))
                      (Let esym (parse-e e)
                           (Prim1 'box (Var esym))))]
[(cons e1 e2)   (let ((esym1 (gensym 'e)))
                 (let ((esym2 (gensym 'e)))
                       (Let esym1 (parse-e e1)
                       (Let esym2 (parse-e e2)
                           (Prim2 'cons (Var esym1) (Var esym2))))))]
```

While this did not solve my problem of thunks not playing well as λ-expressions, compiling `cons` and `box` as `let` resulted in cases such as those cases on lines 97 and 181 working. It's intuitive to think the language is somehow not CBN anymore. However, it still executed correctly on the first and second cases in Table 3. Recall a variable is evaluated by applying its thunk (on no arguments). Therefore, I think the cons cell is really holding two applications of thunks, but I'm not certain.

Attempt 3:

At this point, I felt the issue had to do with difference between compiling thunks and λ-expressions, but even as I write thisI still don't know what it is concretely. I kept running into an issue where variables would not be found in the given environment, presumably because they were free variables. In all liklioood, I was unable to deal with free variables in thunks especially when they because nested in other thunks/λs. I wanted to intuitively separate the way λs and thunks were treated, so I modified `lambdas` to exclude thunks and wrote a function `thunks` that did the opposite, include thunks but not lambdas. Next, I wrote embarrassingly boilerplate code to define thunks similarly to λs, but ignoring anything to do with free variables. This did not work, and I'm still unsure why.

My last ditch effort was to pass a variable `tids` throughout most of `compile.rkt`. `tids` was a list of each thunk's identifier (symbol). My hunch was if I would identify whether or nothing I was compiling a variable inside a thunk, then I could treat it differently.

Conclusion:

Sadly, I couldn't get the compiler working, but this was the most challenging coding I've done in long time. Although it was frustrating, I feel like I learned a lot throwing the kitchen sink at this project. Thanks again for the semester!