CMSC430 Final Project: Source optimizer

Siqi Wang

Introduction

This is a project focus on optimizing the source code, which takes in a program AST and produces an alternative AST that has the same behavior but will execute more efficiently.

All the code of my optimizer is in the file **optimizer.rkt**. The changes that I have made to the original interpreter and compiler are:

```
(define (interp p)                                (define (interp p)
  (match p                                          (match (optimizer p)
    [(Prog ds e)                     -->              [(Prog ds e)
     (interp-env e '() ds)]))                          (interp-env e '() ds)]))
```

```
(define (compile p)                                 (define (compile p)
  (match p                                            (match (optimizer p)
    [(Prog ds e)                                        [(Prog ds e)
     (prog (externs)                 -->                 (prog (externs)
           (Global 'entry)                                     (Global 'entry)
           (Label 'entry)                                      (Label 'entry)
           ...)]))                                             ...)]))
```

Besides, **['err 'err]** for handling error cases is added to the function **interp-env** in the interpreter and **['err (seq (Jmp 'raise_error_align))]** for handling error cases is added to the function **compile-e** in the compiler. My optimizer will produce **'err** directly if an error occurs. So, these modifications will handle the possible **'err** output of my optimizer.

These are the only changes that I have made to the interpreter and compiler. You can easily replicate the following tests by simply adding/removing the call from optimizer in **interp.rkt** and **compile.rkt**.

Optimization

For all **Op1** operations **('add1 / 'sub1 / 'zero? / 'char? / 'integer->char / 'char->integer / 'write-byte / 'eof-object? / 'box / 'car / 'cdr / 'unbox / 'empty? / 'cons? / 'box? / 'vector? / 'vector-length / 'string? / 'string-length)**, they will be computed in optimizer and the output will be in AST that has the same behavior.

Examples:

```
(optimizer (parse '((add1 5))))                          ->  (Prog '() (Int 6))
(optimizer (parse '((integer->char 65))))                -> (Prog '() (Char #\A))
(optimizer (parse '((cdr (cons 1 2)))))                  ->  (Prog '() (Int 2))
(optimizer (parse '((unbox (box 1)))))                   ->  (Prog '() (Int 1))
(optimizer (parse '((char? 8))))                         ->  (Prog '() (Bool #f))
(optimizer (parse '((vector-length (make-vector 3 #f)))))->  (Prog '() (Int 3))
(optimizer (parse '((string? "fred"))))                  ->  (Prog '() (Bool #t))
```

For all **Op2** operations **('+ / '- / '< / '= / 'cons / 'make-vector / 'vector-ref / 'make-string / 'string-ref / 'eq?)**, they will be computed in optimizer and the output will be in AST that has the same behavior.

Example:

```
(optimizer (parse '((vector-ref (make-vector 3 5) 0)))) -> (Prog '() (Int 5))
(optimizer (parse '((+ 1 #t))))                         -> (Prog '() 'err)
(optimizer (parse '((make-vector 3 5))))                -> (Prog '() (Prim2
'make-vector (Int 3) (Int 5)))
```

Other operations **((If Expr Expr Expr) / (Begin Expr Expr) / (Let Id Expr Expr))**, and expressions that include function definition will be computed in optimizer and the output will be in AST that has the same behavior. Expression that includes other operations will be evaluated in the progress for producing the final output.

Example:

```
(optimizer (parse '((if (zero? (add1 2))) 1 2)))) -> (Prog '() (Int 2))

(optimizer (parse '((let ((x 1)) (add1 (if (zero? x) 7 8))))))
-> (Prog '() (Int 9))

(optimizer (parse '((define (map-add1 xs)
                      (if (empty? xs) '()
                        (cons (add1 (car xs)) (map-add1 (cdr xs)))))
                    (map-add1 (cons 1 (cons 2 (cons 3 '()))))))))
->
(Prog '((Defn map-add1 (xs)
        (If (Prim1 'empty? (Var xs)) (Empty)
            (Prim2 cons (Prim1 'add1 (Prim1 'car (Var xs)))
              (App (Var map-add1) ((Prim1 'cdr (Var xs))))))))
    (Prim2 'cons (Int 2) (Prim2 'cons (Int 3) (Prim2 'cons (Int 4) (Empty)))))
```

More complicated tests are all included in **test-runner.rkt**.

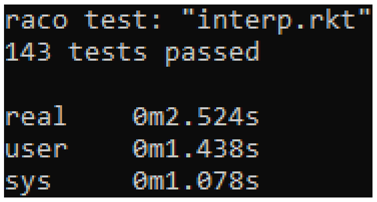
Efficiency

As the examples shown above, the optimizer will pre-evaluate the input AST while keeping the behavior of the original AST. For most of the tests, the lines of assembly code produced by the compiler after optimization will be reduced by nearly 1/2 compared to the original output. Some tests will remain as their original input after optimization because the order of the AST after optimization will cause side effects, which is mentioned in the description of the source optimizer on the class webpage.

Other than counting the reduction in lines of assembly code, the reduction in time is another aspect of showing improvement in code efficiency. Since it is hard to measure the difference of time after/before optimization between one or two tests, I run through all 143 tests **(test-runner.rkt)** in both interpreter and compiler and the difference in time after/before optimization can be easily identified.

Interpreter:

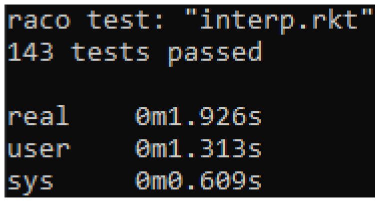
(Before optimization)



```
raco test: "interp.rkt"
143 tests passed

real    0m2.524s
user    0m1.438s
sys     0m1.078s
```

(After optimization)

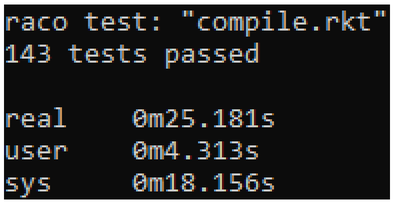


```
raco test: "interp.rkt"
143 tests passed

real    0m1.926s
user    0m1.313s
sys     0m0.609s
```

Compiler:

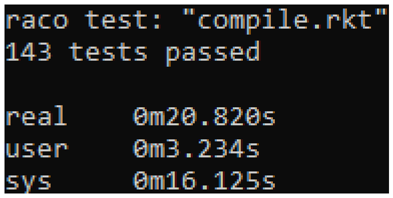
(Before optimization)



```
raco test: "compile.rkt"
143 tests passed

real    0m25.181s
user    0m4.313s
sys     0m18.156s
```

(After optimization)



```
raco test: "compile.rkt"
143 tests passed

real    0m20.820s
user    0m3.234s
sys     0m16.125s
```

These tests can be replicated by adding/removing the optimizer call in **interp.rkt** and **compile.rkt** (as shown above in the introduction) and run command **time raco test interp.rkt** and **time raco test compile.rkt** within the test folder.