By Joshua Cantrell
jjc@cory.berkeley.edu

Analyzing Evaluator

Why Analysis Before Evaluation?

In the evaluator, *eval* has to evaluate an expression before *apply* can be used. The addition of analysis follows a similar line of thought. Everything must be analyzed before it can be evaluated. The major question that should be asked is why not skip analysis or do analysis and evaluation in one single step? The answer is that analysis does preprocessing of expressions once in their lifetime and this processing step can be avoided whenever we decide to evaluate them. In the normal evaluator, we actually do the analysis and evaluation at the same time, but we must always repeat the analysis step, which can become wasteful.

An analogy to this extra step of preprocessing might be building a device or robot that can assemble a particular part in an assembly line. We could build the part by continuously consulting its blueprints and putting it together piece by piece, or we could take the blueprint and build a machine that can quickly assemble the part every time we wanted to assemble a new one. It might take some time to build the machine, but it only needs to be done once, and then we can quickly assemble the particular part when necessary.

What's the Basic Form?

The analysis procedures take one argument, the expression to be analyzed, and return a procedure with one argument, the environment the expression is evaluated within. They have this simple form because to be analyzed, the only information needed is the expression's form. When evaluating an expression, the only thing that changes between evaluations is the environment, so the analyzer's result only requires the environment it is being evaluated within.

Analyzer's Form: `(analyze-something <expression>)`
Analyzer Result's Form: `(analyzed-expression <environment>)`

What Do Analyzers Look Like?

Our book, SICP, does a wonderful job of describing the uncanny resemblance between the analyzing and simple evaluators. Once you know how and why analysis is done ahead of evaluation, the rest should be easy to understand. Pages 393 to 398 cover this material and office hours are always available if you have questions or comments[1].

[1] This may look like a deliberate dodge of the subject, and it is. If I were to fill this document with the information needed to cover this subject, I'd be duplicating that section of SICP.

By Joshua Cantrell
jjc@cory.berkeley.edu

Lazy Evaluator

Normal Order vs. Applicative Order

Scheme evaluates expressions in *applicative order* rather than *normal order*. The excuse given in the first chapter of SICP for using *applicative order* was that delaying evaluation of a procedure's actual parameters (arguments) had a tendency of making redundant evaluations in the future. The example was:

```
> (define (square x) (* x x))
> (square (+ 5 4))
```

The result of evaluating this in *normal order* produced the following series of steps:

```
(square (+ 5 4))
(* (+ 5 4) (+ 5 4))
(* 9 (+ 5 4))
(* 9 9)
81
```

The result of evaluating this in *applicative order* produced the following series of steps:

```
(square (+ 5 4))
(square 9)
(* 9 9)
81
```

Notice how we had to evaluate `(+ 5 4)` twice in our *normal order* evaluation, but only once in our *applicative order* evaluation. You may be asking why we'd want to take the penalty of evaluating more redundant expressions just to delay the evaluation of an argument. The reason is that it can be useful in some cases. Just like when we were dealing with streams and delaying evaluation was useful, sometimes delaying the evaluation of a procedure's actual parameters is also useful.

Thunk but didn't do.

In the *lazy evaluator* we will be delaying expressions that are arguments to procedures by making them *thunk*s. *Thunk*s in the *lazy evaluator* are different than those used in our stream implementation because of one special feature: when forced, they are evaluated in the same environment they were delayed in. In our stream implementation, we delayed our expressions by making them a body of a procedure. The result of this was that when we evaluated the expression (by applying the procedure to no arguments), it was technically in a different environment than the one it was delayed in. The frame the expression was evaluated in pointed to the environment it was delayed in, but was not the same environment the expression was delayed in. In this case the environment the expression is delayed in is saved with the *thunk* and used later when the expression is finally evaluated.

The term *thunk* arose because of the analyzing evaluator. The expression could be analyzed ahead of time, and therefore was "thunk" about, but hadn't been fully evaluated. It

By Joshua Cantrell
jjc@cory.berkeley.edu

would later be a "thunk" about expression that was finally evaluated using the environment it was first "thunk" about in.

What elements make up a *thunk*?

There are two elements that make up a *thunk*. First, you need the delayed expression which could be pre-analyzed or not. Second, you need a pointer to the environment (or current frame) that the *thunk* was made in. The book defines a *thunk* to have the following structure:

```
(thunk <delayed-expression> <environment>)
```

What needs to be *thunk*ed?

There are arguments that need to be *thunk*ed, and others that don't. The difference is described as *strict* versus *non-strict*.

non-strict - The body of a procedure is entered before an argument has been evaluated.
strict - The argument is evaluated before the body of the procedure is entered.

Primitive procedures are *strict*, so their arguments must be evaluated before they are applied. Compound procedure are *non-strict*, so their arguments are *thunk*ed and then they are applied.

Where do we un*thunk*?

We need to un*thunk* an expression in three locations. The first is the un*thunk*ing of all arguments to *strict* procedures (in our case, only primitive procedures). The second is un*thunk*ing the predicate of an *if* statement when it is evaluated (because the *#t* or *#f* value must be returned before we know whether to choose the consequent or alternative expression). The third is un*thunk*ing the result that is returned to the user (obviously, the user doesn't want an unevaluated expression if one is returned).

Lazy Lists vs. Streams

Before we made up streams to get around the fact that everything is evaluated in *applicative order*. Now with *normal order* evaluation, everything is delayed when it's used as the argument to a compound procedure. To make a pair with two *thunk*ed values, all that needs to be done is make our own *cons*, *car*, and *cdr* imitators.

```
(define (cons x y) (lambda (m) (m x y)))
(define (car z) (z (lambda (p q) p)))
(define (cdr z) (z (lambda (p q) q)))
```

Note that we could also have used message passing because that would have lead to the values being *thunk*ed.

The main difference between our *lazy list*s and *stream*s is that only the *cdr* part of a *stream* was delayed, while both the *car* and *cdr* part of a *lazy list* is delayed. This creates a super lazy stream. Page 410 of SICP shows some of the equivalents of *stream* procedures that are for *lazy list*s.

By Joshua Cantrell
jjc@cory.berkeley.edu

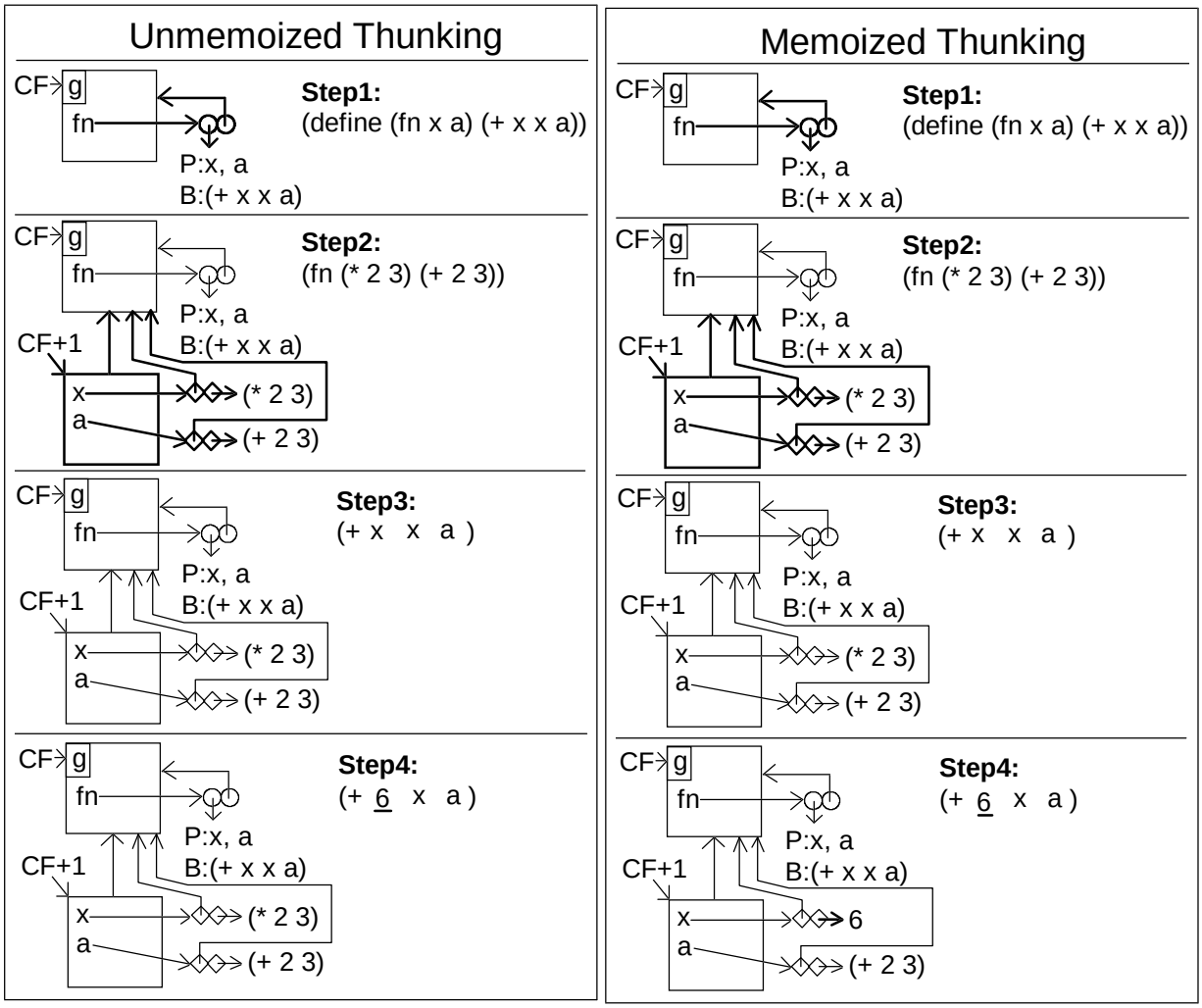
Memoizing Thunks vs. Not Memoizing Thunks

When dealing with functions, we decided that we could memoize the value a function returned given particular arguments because functions (by definition) will always return the same value given the same input. Then when we approached streams, since they were also functional, we found out we could use the same method of memoizations to increase the efficiency and speed of streams. Now, when dealing with the *lazy evaluator*, we have the choice to memoize a *thunk*'s value after it has been finally evaluated, and use that value everywhere it appears instead of reevaluating it.

Memoization of *thunk*s is unlike what was done with the memoization of functions because a *thunk* may look the same as another *thunk*, but would still be evaluated seperately because each *thunk* would still be a unique entity. One way to visualize a *thunk* is as an object that points to the expression needed to be evaluted, or some result of an expression (what's created when memoization occures after the first evaluation of the *thunk*ed expression).

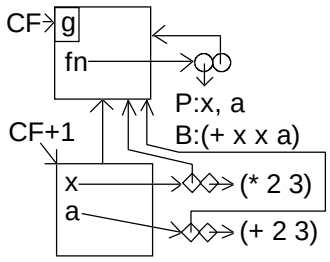
As an example here are steps for the drawing of an environment diagram with memoized and unmemoized *thunk*s. Notice how the unmemoized *thunk*s are never replaced with the resulting value, whereas the memoized *thunk*s are. Also notice how each *thunk*ed expression makes its own *thunk* object. The thunks also point to the environment they are evaluated within.

```
> (define (fn x a) (+ x x a))
> (fn (* 2 3) (+ 2 3))
```

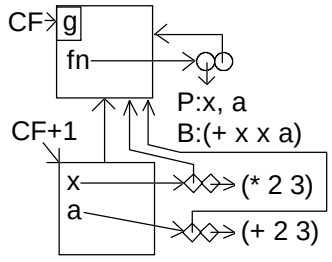


By Joshua Cantrell
jjc@cory.berkeley.edu

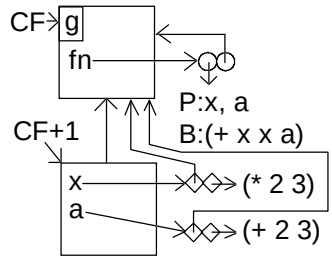
Unmemoized Thunking



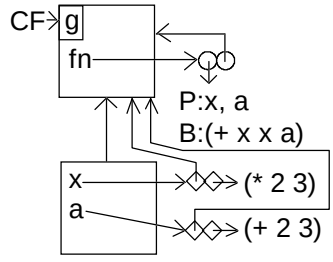
Step5:
(+ 6 6 a)



Step6:
(+ 6 6 5)

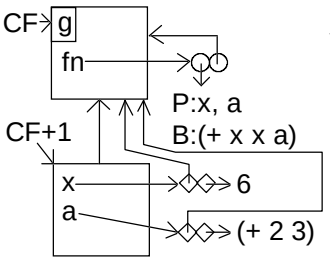


Step7:
17

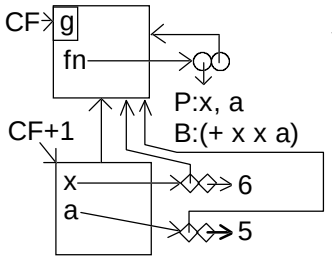


Step8:
17

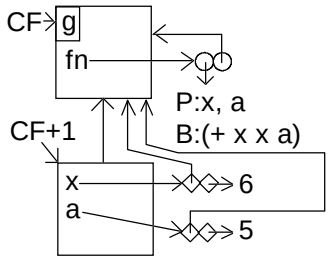
Memoized Thunking



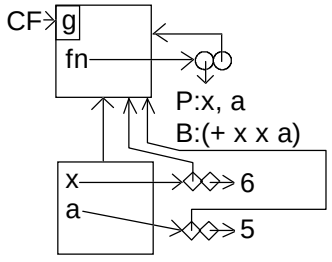
Step5:
(+ 6 6 a)



Step6:
(+ 6 6 5)



Step7:
17



Step8:
17