CPSC 311: Definition of Programming Languages: Lazy evaluation

21-lazy

Joshua Dunfield
University of British Columbia

November 13, 2015

1 Evaluation strategies: review and update

1.1 Review

Earlier in the course, we came up with two strategies for evaluating function application (app $e1$ $e2$): the *expression strategy*, in which the function argument $e2$ is substituted for x in the body of (lam x eB), and the *value strategy*, in which the function argument $e2$ is evaluated immediately and the resulting value $v2$ is substituted for x in eB.

Under the expression strategy, we evaluate $e2$ as many times as (id x) is evaluated; under the value strategy, we evaluate $e2$ exactly once. The value strategy is usually more efficient than the expression strategy, but the expression strategy is more efficient if (id x) is not evaluated. For example, in

$$\big(\mathsf{app}\ (\mathsf{lam}\ x\ (\mathsf{num}\ 0))\ e2\big)$$

there is no (id x) in the body of the lam, so (under the expression strategy) the argument $e2$ will never be evaluated.

Exercise 1. Give a slightly larger (and hopefully less contrived) example of a function application for which the expression strategy is more efficient than the value strategy. (Hint: apply a function of *two* arguments, that is, (lam x (lam y ...)).)

1.2 Update for environment-based evaluation

We have mostly used the value strategy, and we stayed with that strategy in developing the environment-based evaluation rule for app, Env-app; for clarity, we'll now call that rule Env-app-value:

$$\frac{env \vdash e1 \Downarrow (\mathsf{clo}\ env_{old}\ (\mathsf{lam}\ x\ eB)) \qquad env \vdash e2 \Downarrow v2 \qquad x{=}v2, env_{old} \vdash eB \Downarrow v}{env \vdash (\mathsf{app}\ e1\ e2) \Downarrow v}\ \text{Env-app-value}$$

To facilitate experimenting, I'm leaving the value-strategy application app alone, but adding an expression-strategy application app-expr.

§1 Evaluation strategies: review and update

To implement the expression strategy, we might try

$$\frac{env \vdash e1 \Downarrow (\text{clo } env_{\text{old}}\ (\text{lam x } eB)) \qquad x{=}(\text{clo } env\ e2), env_{\text{old}} \vdash eB \Downarrow v}{env \vdash (\text{app-expr } e1\ e2) \Downarrow v}\ \text{??Env-app-expr}$$

As we did for lam, we need to save the current environment so that when (id x) is evaluated, we can evaluate $e2$ under env rather than some later environment.

Using the same rules as before for id and clo won't quite work:

$$\frac{lookup(env, x) = e \qquad e \Downarrow v}{env \vdash (\text{id x}) \Downarrow v}\ \text{Env-id} \qquad \frac{}{env \vdash (\text{clo } env_{\text{old}}\ e) \Downarrow (\text{clo } env_{\text{old}}\ e)}\ \text{Env-clo}$$

Let's see what happens when we try to evaluate

(app-expr (lam x (add (id x) (id x))) (add (num 1) (num 2)))

in an empty environment:

1. We evaluate (lam x (add (id x) (id x))) to (clo ∅ (lam x (add (id x) (id x)))).
2. Instead of evaluating (add (num 1) (num 2)), we extend the environment with

   x=(clo ∅ (add (num 1) (num 2)))

   and evaluate the body (add (id x) (id x)) under that environment.
3. Following Env-id, we evaluate the first (id x) by looking up x in the environment, which gives us our e:

   $$e \;=\; (\text{clo } \emptyset\ (\text{add } (\text{num } 1)\ (\text{num } 2)))$$

   Now, following Env-id, we evaluate e. But a clo evaluates to itself, so we get

   $$v \;=\; e \;=\; (\text{clo } \emptyset\ (\text{add } (\text{num } 1)\ (\text{num } 2)))$$

   and return that v in the conclusion.

   Unfortunately, this e isn't a num, so `interp-env` raises an error!

There are at least three approaches to fixing this problem:

- Inspired by Env-app[-value], we could add rules "Env-add-clo-..." to handle the cases in which one, or both, of the values is a clo. This requires *three* additional rules—and we would need to do something similar for *every* feature that "eliminates" (uses) a value. For example, the rule for pair-case would need to handle the case in which the scrutinee is not a pair, but a clo.
- We could change the rules for id and/or clo somehow.

  This might work, I'm not sure, but (after my misadventure with recursive closures) I'm reluctant to try, because I don't want to risk breaking application. If it did work, it would probably require fewer additional rules than the approach above.

- We could introduce another kind of closure. This allows us to define the semantics of that new closure separately, without affecting the semantics of clo.

I'm taking the last approach, partly because it seemed to work for recursive closures.

For historical reasons, the new kind of closure will be called a *thunk*.

```
(define-type E
  [num (n number?)]
  [binop (op Op?) (lhs E?) (rhs E?)]

  [with (name symbol?) (named-expr E?) (body E?)]
  [id (name symbol?)]

  [lam (name symbol?) (body E?)]

  [clo (env Env?) (e E?)]                ; not in concrete syntax
  [clo-rec (box-env box?) (e E?)]        ; not in concrete syntax
  [thk (env Env?) (e E?)]                ; not in concrete syntax

  [app (function E?) (argument E?)]
  [app-expr (function E?) (argument E?)]

  [rec (name symbol?) (body E?)]
```

The rule for evaluating thk turns out to be almost the same as the rule for clo-rec:

$$\frac{env \vdash e1 \Downarrow (\text{clo } env_{old}\ (\text{lam x } eB)) \qquad x{=}(\text{thk } env\ e2), env_{old} \vdash eB \Downarrow v}{env \vdash (\text{app-expr } e1\ e2) \Downarrow v}\ \text{Env-app-expr}$$

$$\frac{env_{old} \vdash e \Downarrow v}{env \vdash (\text{clo-rec } env_{old}\ e) \Downarrow v}\ \text{Env-clo-rec} \qquad \frac{env_{old} \vdash e \Downarrow v}{env \vdash (\text{thk } env_{old}\ e) \Downarrow v}\ \text{Env-thk}$$

The rules Env-app-expr (for app-expr) and Env-thk are implemented in `env-thunks.rkt`.

Examples:

```
; for app-expr, do we evaluate the argument twice?
(unparse (interp (parse '{app-expr {lam x {+ x x}} {+ 1 2}})))

; does it work when the argument is a lam?
(unparse (interp (parse '{app-expr {lam f {app f 5}} {lam y {+ y y}}})))

; does it work when the argument has a free variable (z)?
(unparse (interp (parse '{with {z 100}
                               {app-expr {lam f {app f 5}}
                                         {lam y {+ y z}}}})))

; are we still doing lexical scope?
```

```
(unparse (interp (parse '{with {z 100}
                               {app-expr {lam f {with {z 444} {app f 5}}}
                                         {lam y {+ y z}}}})))
```

2 Lazy evaluation

Environments make it possible to implement a third evaluation strategy, which I think is pretty clearly better than the expression strategy. Whether it's better than the value strategy is unclear.

It's useful to pause and relate my terminology in this course to terminology that you may come across elsewhere. I invented my own terminology because I think it's less confusing, but you should be aware of the more common usage:

invention	**CPSC 311 name**	**"formal" names**	**"popular" names**	**other names**
1950s	value strategy	call-by-value, CBV	eager evaluation, strict evaluation	[applicative order]
1960	expression strategy	call-by-name, CBN	by name	[normal order]
1971–76	lazy evaluation	call-by-need	lazy evaluation	

Brackets, e.g. "[applicative order]", indicate that there is no consensus that those terms are being used accurately, and that many people will (probably correctly) object and reserve those terms for different concepts. I mention them because you may come across them, and they aren't *entirely* wrong: applicative order is *more like* the value strategy than it is like the expression strategy.

Also, people often say "evaluation order" rather than "evaluation strategy". But I prefer "strategy", at least in 311, because it's not just the *order* in which expressions are evaluated: it's also whether they're evaluated at all.

2.1 Overview

From a distance, lazy evaluation looks like the expression strategy:

- function arguments are not evaluated immediately;
- function arguments are only evaluated when used.

The difference from the expression strategy is in what happens when the argument is evaluated, *if* it is evaluated:

- The expression strategy evaluates the argument, but doesn't remember the result. So if it sees (id x) again, it evaluates the argument again.
- Lazy evaluation remembers the result of evaluating the argument. If it sees (id x) a second (or third...) time, it returns the result without evaluating it again.

We can't use the environment to remember the result, however, because the environment is only passed *up* the evaluation derivation, not "threaded through". But earlier, we added support for mutable references (boxes), which live in a store that *is* threaded through! So we can use the store to remember our results.

(By the way, this is roughly how lazy evaluation actually works in languages that use it, like Haskell.)

2.2 Rules

We want to put the result of evaluating an argument in the store; since the store is what survives leaving a lexical scope, we also need to remember in the store *whether* we have evaluated that argument. The environment will *refer* to the store, using a location.

I left out the store from the above rules, so let's put that in Env-app-expr and then rewrite Env-app-expr to be lazy.

$$\frac{env; S \vdash e1 \Downarrow (\text{clo } env_{old}\ (\text{lam x } eB)); S1 \qquad x{=}(\text{thk } env\ e2), env_{old}; S1 \vdash eB \Downarrow v; S2}{env; S \vdash (\text{app-expr } e1\ e2) \Downarrow v; S2}\ \text{Env-app-expr}$$

To avoid using the name thk for both the expression strategy and lazy evaluation, I'll use lazythk.

$$\frac{env; S \vdash e1 \Downarrow (\text{clo } env_{old}\ (\text{lam x } eB)); S1 \qquad x{=}(\text{lazy-ptr } \ell), env_{old}; \ell{\triangleright}(\text{lazy-thk } env\ e2), S1 \vdash eB \Downarrow v; S2}{env; S \vdash (\text{app-lazy } e1\ e2) \Downarrow v; S2}\ \text{Env-app-lazy}$$

In the second premise of Env-app-lazy:

1. We bind x to (lazy-ptr ℓ). Since the environment isn't threaded through, x will *always* be bound to (lazy-ptr ℓ) for the entire time that x is in scope.
2. We extend the store with a new location ℓ containing (lazy-thk env $e2$).

Note that we haven't evaluated $e2$ yet—and if evaluating eB does not evaluate (id x), we never will.

When we evaluate (id x), we will look it up (Env-id) and evaluate what we find. The environment (not the store!) has x=(lazy-ptr ℓ), so we find (lazy-ptr ℓ) and evaluate that.

We need a rule for the case where we haven't evaluated the argument yet. In that case, looking up ℓ in the store will give a lazythk. We evaluate $e2$ under the environment that Env-app-lazy saved, resulting in a value v, and use *update-loc* to replace ℓ with v:

$$\frac{\textit{lookup-loc}(S, \ell) = (\text{lazy-thk } env_{arg}\ e2) \qquad env_{arg}; S \vdash e2 \Downarrow v; S1 \qquad \textit{update-loc}(S1, \ell, v) = S2}{env; S \vdash (\text{lazy-ptr } \ell) \Downarrow v; S2}\ \text{Env-lazy-ptr}$$

We also need a rule for the case where we already applied Env-lazy-ptr to ℓ. In that case, looking up ℓ in the store will *not* give a lazythk, but some value—the v that we got while applying Env-lazy-ptr.

$$\frac{\textit{lookup-loc}(S, \ell) = v \qquad v \neq (\text{lazy-thk } \cdots\ \cdots)}{env; S \vdash (\text{lazy-ptr } \ell) \Downarrow v; S}\ \text{Env-lazy-ptr-done}$$

```
(unparse (interp (parse '{app-lazy {lam x {+ x x}} {- 10 1}})))
```

2.3 Ideology

Evaluation strategy, like typing, is one of the most enduring controversies in programming language design. The controversy began the moment there was more than one evaluation order:

> The first call-by-name language, Algol 60, also supported call-by-value. It seems that call-by-value was the language committee's preferred default, but Peter Naur, the editor of the Algol 60 report, independently reversed that decision—which he said was merely one of a "few matters of detail". A committee member, F.L. Bauer, said this showed that Naur "had absorbed the Holy Ghost after the Paris meeting... there was nothing one could do... it was to be swallowed for the sake of loyalty." (From Dunfield (2015), "Elaborating evaluation-order polymorphism"; quotations from Wexelblat (1981), *History of Programming Languages I*.)

(There was also some argument about whether Naur had independently decided to include recursion in Algol-60, but my reading is that he didn't do that—the committee had agreed to support recursion, but may have had arguments over details.)

Later developments continued to be full of ideology. Lazy evaluation (under the name call-by-need) was introduced in a 1971 PhD thesis, and first implemented (twice, mostly independently, I believe) in 1976. One of the 1976 papers has the rather opinionated title "CONS Should Not Evaluate its Arguments", only softened slightly in the paper itself: "we have uncovered a critical class of elementary functions which probably should never be treated as strict: the functions which allocate or *cons*truct data structures." I suspect that not all ML programmers would agree. Nor would users (at least, designers) of Lisp, the language used in the 1976 paper: neither Lisp nor its descendants Scheme and Racket are lazy.

These controversies are (partly) grounded in legitimate disagreements about design tradeoffs between the value strategy (eager evaluation) and lazy evaluation:

- The value strategy trades speed in one, possibly uncommon case—the case where the argument isn't used—for simplicity.
- Lazy evaluation trades simplicity for speed, but also trades space for speed: the many thunks that have to be created take up space (and slow down garbage collection) even if they are never evaluated. Reasoning about how much space is used is difficult; for example, I believe that the Haskell community relies on *space profilers* to debug "space leaks" that result from thunks being built that are never needed.
- Lazy evaluation sometimes trades actual improvement for apparent improvement: if the expression whose evaluation is being avoided is simple, it would be faster to evaluate it without creating a thunk—*even if the argument is never used*.

There are certainly cases where lazy evaluation is superior, but opponents of lazy evaluation argue that these cases can be handled by explicit programmer-controlled laziness instead. That is, laziness should be an option that must be asked for explicitly, rather than the default.

2.4 Function application vs. the whole language

The rules we've developed add lazy function application, without changing any other language constructs. Languages with built-in laziness usually don't stop there. For example, Haskell is entirely lazy: if you add two expressions with +, the addition won't be performed unless the result is "demanded" (such as by printing the value to the user).