Lazy Evaluation or Call-by-Name

Our description of function call (so far) has involved

- simplification (evaluation/calculation) of the "operator" expression e_1 to a function f (or in the operational semantics, to a data structure representing a function, namely a "value closure" of the form $\langle\langle \lambda x . e', \gamma' \rangle\rangle$)
- simplification (evaluation/calculation) of the "argument" expression e_2 to a value/ canonical answer $a_2 \in Ans$ (where the set of answers includes "value-closures")
- augmenting the table γ' saved in the closure with the binding of result a_2 of the argument expression e_2 to the *formal parameter* x, and then evaluating/calculating the "body" e' of the "operator" closure to yield an answer a.

Now let us consider the possibility of expressions not terminating (always possible in a Turing-complete language). Of course, (in the first step) if the evaluation/calculation of the operator expression e_1 did not terminate, the function call does not terminate. And in the third step, if the evaluation/calculation of the "body" expression e' did not terminate, then we would get no resulting value/answer.

However, suppose (in the first step) the evaluation/calculation of the operator expression e_1 terminated as desired with a function/"value closure", but (in the second step), the evaluation/calculation of the argument expression e_2 did not terminate, then what should happen. In most of the languages we may have used so far, the function call does not terminate, and no value/answer is returned.

However, there is no reason in mathematics that this must be so. Consider the "constant function" $\lambda x.3$, which given any argument, always returns the answer 3. Suppose we applied this function to an expression whose evaluation does not terminate (let us denote such an expression as Ω; we shall see that we can easily write such expressions in any Turing complete language, just as easily as writing an infinite loop in Python or Java). What answer should the calculation of $((\lambda x.3)\ \Omega)$ yield? Or what is the mathematical meaning of $((\lambda x.3)\ \Omega)$ in any given environment? If indeed we meant it to be the mathematical "constant function", then the answer should be 3, irrespective of what input was given *including when the input is not defined*! But neither our denotational semantics nor certainly do our operational rules say so. In fact, the latter clearly will *not* specify any answer to be returned.

So we need to be more careful in defining the meaning of evaluation and calculation, and make explicit what semantics we wish to follow. If we follow the implementational tradition from Fortran, C, C++ etc., we *eagerly* evaluate the arguments to a function. If we wish to be closer to mathematics, as is the case in languages such as Haskell, we will be *lazy* about evaluating the argument…because it may not terminate, but may not be relevant (i.e., not needed) in the body of the function.

Note that our notion of function space only talks about what the behaviour of the function f should be when given a defined argument a. Interestingly, the definition of function space as such does not need to change, but we may need to clarify which function we mean when the argument is undefined. This in fact required considerable technical inventiveness on various mathematicians/logicians/computer scientists such as Dana Scott, Christopher Strachey and others to identify the appropriate mathematical domains for appropriate treatments of nontermination, and the meaning of (higher-order) functions in the presence of recursion and nontermination.

Strict and Non-strict Functions

We call a function *strict* if it is undefined when its argument is undefined. Here we are equating the notion of a value being undefined to nontermination. (Note that this may not in general be the case). Otherwise, if a function's result is defined even on being given an undefined input argument, it is non-strict. There is, of course, a spectrum of functions which are strict on certain arguments and non-strict on others — for example, the function described by $\lambda x\,.\,\lambda y\,.\,x$ is strict in its first argument, namely x, whereas it is non-strict in its second argument, namely y. "Dually", the function described by $\lambda x\,.\,\lambda y\,.\,y$ is non-strict in its first argument, namely x, whereas it is strict in its second argument, namely y. The identity function $\lambda x\,.\,x$ is strict in its (only) argument. (We can quibble about whether the "constantly undefined" function $\lambda x\,.\,\Omega$ is strict or not!)

Digression: Further, in the same sense as we have encountered in calculus, we have a notion of approximation — and an undefined value is treated as an approximation of a defined value. The meaning of recursive functions (or loops) will be treated as the culmination (a "limit") of a series of better and better approximations in a (complete) partial order … much the same way that a real number is considered the limit of a series of rationals that are increasingly closer to it.

So let us re-examine the denotational definition of function abstractions and function call:

$$eval[\![\underline{\lambda x\,.\,e}\,]\!]\,\rho = \quad f \in [A \rightharpoonup B]$$
where $f(a) = eval[\![\,\underline{e}\,]\!](\rho[x \mapsto a])$ for each $a \in A$

We invent an "undefined value", commonly written as $\bot$, and affix it to any set A to form a partial-ordered structure $(A_\bot, \sqsubseteq)$ such that that has $\bot \sqsubseteq a \in A_\bot$.

In a "non-strict" interpretation, even for input $\bot$, $f(\bot) = eval[\![\,\underline{e}\,]\!](\rho[x \mapsto \bot])$ could be defined, especially if x does not appear free in expression e, due to the Relevance Lemma.

However, in a "strict" interpretation, we would require that $eval[\![\underline{\lambda x\,.\,e}\,]\!]\,\rho$ is a *strict* function, i.e., an element of a (sub)function space, usually denoted as $f \in [A_\bot \rightharpoonup B_\bot]_\bot$. (Theoretically, there also can be hybrids such as $[A_\bot \rightharpoonup B]_\bot$, $[A \rightharpoonup B_\bot]$, $[A \rightharpoonup B]_\bot$, etc. though only very few are useful.)

How about function call? Interestingly, we *can* write leave the semantic equation unchanged, but will tweak what we mean by the mathematical elements f and a

$$eval[\![\,\underline{(e_1\ e_2)}\,]\!]\,\rho = f(a)$$
where $f = eval[\![\underline{e_1}]\!]\rho$ and $a = eval[\![\underline{e_2}]\!]\rho$

but we decide whether a may be $\bot$ (or not), and whether $f \in [A \rightharpoonup B]$ or $f \in [A_\bot \rightharpoonup B_\bot]_\bot$.

Now all of this is really the subject for a more advanced course on programming languages, but it is worthwhile to know that these difficult questions required being resolved for one to have a good sense of what are computable functions, and how to properly give meaning to programs.

Lazy or Call by Name Functional Languages

We focus on the sublanguage

$$e \in Exp \ ::= \ \ldots \mid x \mid \lambda x \,.\, e \ \mid \ (e_1 \ e_2)$$

to explain and illustrate the differences.
Note that the language hasn't changed, not does the possible OCaml representation of its abstract syntax.

```
type exp = V of string | Abs of string * exp | App of exp * exp ;;
```

Even more importantly (for you), the typing rules that we have given *do not change* irrespective of which semantics we choose — since the typing only placed constraints on what type a results should belong to (and was silent about what happens when an expression does not terminate). [Note: there are languages with type systems that guarantee termination. Those are beyond the scope of this course.]

$$\tau, \tau_1, \tau_2 \in Typ \ ::= \ \ldots \mid \tau_1 \rightarrow \tau_2$$

$$\textbf{(AbsT)}\ \frac{\Gamma[\underline{x} : \tau_1] \vdash \underline{e} : \tau_2}{\Gamma \vdash \underline{\lambda x \,.\, e} : \tau_1 \rightarrow \tau_2} \qquad\qquad \textbf{(AppT)}\ \frac{\Gamma \vdash \underline{e_1} : \tau_1 \rightarrow \tau_2 \qquad\qquad \Gamma \vdash \underline{e_2} : \tau_1}{\Gamma \vdash \underline{(e_1 \ e_2)} : \tau_2}$$

Operational Semantics (Lazy Version)

We say that the construct in a language treats a component lazily if the evaluation of the construct does not *require* the computation of that component. Otherwise it is eager in evaluating/computing/calculating that component. For example, the if-then-else conditional construct is eager in evaluating its test subexpression e_0 but lazy in evaluating its then and else branches e_1 and e_2, evaluating only the one that is found to be necessary by the computation of the test e_0. This is a good example of why the conditional is a construct in the language, and not something that can be coded as a function in an eager language such as OCaml. *However, it is not possible to determine in an arbitrary program whether a particular subexpression requires evaluation or not.* (Of course, for particular programs/function abstractions, we may be able to do so... but not in general, for arbitrary inputs....).

[Note that every eager language is bound to have such (partially) lazy constructs. Conversely, every lazy language must have at least one eager construct, otherwise there is nothing to kick-start any computation, and every expression will just sit and not be evaluated. Remember this fact not only when you read operational semantic rules, but also when you go on to write production code.]

Closures and Tables Revisited

Recall we introduced closures to correctly implement the lexical scoping discipline, so that we could correctly bind all the (apparently) free variables which appear in a function body to their meanings in the prevailing environment. Now in lazy evaluation of a function, we may not know at call time whether we will require the argument to be evaluated or not. So we defer the evaluation. If it requires evaluation, we do so. But we must do so in the environment where it was passed in as an argument, not in the currently prevailing environment. If it doesn't require evaluation, then we are lucky not to have had to evaluate it. This requires creating a *closure* for each argument to a function. In general this is expensive. And note that a closure now can contain an arbitrary expression e (and not only abstractions $\lambda x . e$).

So now

$$Clos = Exp \times Table$$

and

$$Table = \mathcal{X} \rightharpoonup_{fin} Clos$$

The so-called "value closures" are still represented as $\langle\langle \lambda x . e,\ \gamma \rangle\rangle$ in the subset $VClos = \{\lambda x . e \ \mid\ x \in \mathcal{X}, e \in Exp\} \times Table$ of $Clos$. $VClos$ forms a subset of canonical answers Ans, the results of calculation. Note that $Clos$, which may represent incompletely computed expressions need not (is not) contained in Ans.

We are now in a position to write the correct Big-step rules for Lazy Evaluation for variables, function abstractions and application:

$$(\textbf{CalcVarClosL})\ \frac{\gamma' \vdash \underline{e'} \Longrightarrow_{cbn} \underline{a}}{\gamma \vdash \underline{x} \Longrightarrow_{cbn} \underline{a}} \text{ when } \gamma(x) = \langle\langle e', \gamma' \rangle\rangle$$

Since in a table now a variable may be mapped to a general closure containing a possibly unevaluated expression e', that expression has to be first evaluated (in the context of the saved environment, i.e., table γ') to an answer a — which is the result of calculating the variable.

$$(\textbf{CalcAbsL})\ \frac{}{\gamma \vdash \underline{\lambda x . e} \Longrightarrow_{cbn} \langle\langle \lambda x . e,\ \gamma \rangle\rangle}$$

That is, an abstraction still calculates to a closure (which is a canonical answer) with the call-time table packed into the (value)-closure.

$$(\textbf{CalcAppL})\ \frac{\gamma \vdash \underline{e_1} \Longrightarrow_{cbn} \langle\langle \lambda x . e',\ \gamma' \rangle\rangle \qquad \gamma'\ [\underline{x} \mapsto \langle\langle e_2, \gamma \rangle\rangle] \vdash \underline{e'} \Longrightarrow_{cbn} \underline{a}}{\gamma \vdash \underline{(e_1\ e_2)} \Longrightarrow_{cbn} \underline{a}}$$

The operational semantics rule (**CalcAppL**) details how the result of a function call $(e_1\ e_2)$ is calculated: First we calculate the answer for the "operator" expression e_1, which being a function, should result in a closure. Note that this answer is a "value closure" of the form $\langle\langle \lambda x . e',\ \gamma' \rangle\rangle$ where the expression component is an abstraction of the form $\lambda x . e'$, where x is the formal parameter, and e' the body of the function. Note also that the table packed into the closure *may* be different from the table γ with respect to which we calculated the answer (for example, it may be the result for looking up the table for a named function, which was created in another environment), and hence we indicate this possibly different table as γ'. In lazy evaluation, we do *not* evaluate the "argument" expression e_2, but instead pack it into a closure with the same call-time environment, namely table γ. This closure is bound to (in this call-by-name or lazy discipline)to the formal parameter x. Now the body e' of the function closure is evaluated, with respect to the environment γ' saved in the closure, augmented with the binding $\underline{x} \mapsto \langle\langle e_2, \gamma \rangle\rangle$. The answer a obtained from this evaluation of the function body is returned as the answer of the function call. If ever the variable x needs to be evaluated, we calculate the result of the closure to which it is bound, following rule (**CalcVarClosL**).

Exercise: Write a function *unpack* from closures to expressions, that recursively unpacks a closure into an expression, by substituting for a variable appearing in the expression component of a closure the unpacking of the closure to which it is bound in the table of the closure.

Once you have done so, you may wish to try your hand at the following:

The Type Preservation theorem also continues to hold!

Exercise: Prove the new induction cases in the Type Preservation Theorem.

Exercise: Prove the new induction cases in the Soundness Theorem.

Exercise: Prove the new induction cases in the Completeness Theorem.

Alternative Formulation: Big-step as closure evaluation

Again, we can reformulate the call-by-name (lazy) big-step natural semantics as a transition between closures, which returns an answer (which may be a "value closure") Here is this alternative formulation of these rules, presented when all answers are closures. So the semantics transforms closures in $Clos$ into a subset of "value closures" in $VClos$.

(**CalcVarL'**) $$\frac{\gamma(x) \Longrightarrow_{cbn-clos} a}{\langle\langle \underline{x}, \gamma \rangle\rangle \Longrightarrow_{cbn-clos} a}$$

where the variable x is looked up in table γ, and we obtain a closure $\gamma(x) = \langle\langle e', \gamma' \rangle\rangle$.

(**CalcAbsL'**) $$\frac{}{\langle\langle \lambda x \,.\, e,\ \gamma \rangle\rangle \Longrightarrow_{cbn-clos} \langle\langle \lambda x \,.\, e,\ \gamma \rangle\rangle}$$

is a trivial rule.

Application is rewritten in terms of closure simplification as:

(**CalcApp'**) $$\frac{\langle\langle \underline{e_1},\ \gamma \rangle\rangle \Longrightarrow_{cbn-clos} \langle\langle \lambda x \,.\, e',\ \gamma' \rangle\rangle \qquad \langle\langle \underline{e'},\ \gamma'\,[\underline{x} \mapsto \langle\langle \underline{e_2},\ \gamma \rangle\rangle] \rangle\rangle \Longrightarrow_{cbn-clos} a}{\langle\langle \underline{(e_1\ e_2)},\ \gamma \rangle\rangle \Longrightarrow_{cbn-clos} a}$$

The Type Preservation theorem continues to hold for this formulation (as do the other theorems).

Exercise: Prove the new induction cases in the Type Preservation Theorem.
Caution: One has to be quite careful in this proof, in stating and using the IH.

Compilation and Stack Machine Execution

We now use the notion of closures to define a stack-based evaluator for the *call-by-name* pure λ-calculus, where the argument to a function is *not* evaluated at the time of function call, but only when it is required when evaluating the body of the function. In order to correctly record the bindings of the free variables in the argument expression, closures are used.

The *Krivine* Abstract Machine configuration consists of a "focus" closure and a *stack* of closures corresponding to *unevaluated arguments*. Its configurations are given by $KConf = Clos \times (Clos \text{ stack})$

The machine is specified using the following one-step transitions.

- (**KrOp**) Stack the unevaluated argument e_2 (with table γ packed into a closure) and focus on the operator e_1

  $(\langle\langle(e_1\ e_2), \gamma\rangle\rangle\ , s\ )$ => $(\langle\langle e_1, \gamma\rangle\rangle, \langle\langle e_2, \gamma\rangle\rangle :: s\ )$

- (**KrVar**) Look up the variable x in the table γ, and focus on the closure $\gamma(x)$ to which x is bound.

  $(\langle\langle x, \gamma\rangle\rangle\ , s\ )$ => $(\ \gamma(x)\ ,\ s\ )$

- (**KrApp**) Pop the argument closure cl from the stack and bind it to the formal parameter x, and focus on the body e' of the λ-abstraction with the augmented table.

  $(\langle\langle \lambda x . e', \gamma\rangle\rangle,\ cl :: s)$ => $(\langle e', \gamma[x \mapsto cl]\rangle,\ s)$

Run these transitions whenever possible, starting with a given closure.

Exercise: Under what conditions does the machine halt? When does it do so in a desired terminal state with an answer? When can it get stuck in an undesirable state?

Exercise: Define a tail-recursive OCaml function `krivine: clos * (clos list) -> clos * (clos list)` which implements the execution of the machine till it cannot make any further steps.

Exercise: Define a Call-by-Name evaluator as an OCaml function `cbn: exp -> exp` which takes a given closed pure λ-calculus expression e, forms a closure with the empty table, runs the `krivine` machine and using `unpack` on the resulting focus closure when the machine halts, returns an answer.