Lazy Evaluation

In contrast, a lazy language attempts to avoid “unnecessary” computation by deferring evaluation of function arguments and data structure components, e.g., \( \lambda x : \text{nat}.x + x(x) \)

In the call-by-name evaluation strategy, arguments and components are only evaluated when (and every time) they are actually used.

In the call-by-need evaluation strategy, arguments and components are only evaluated the first time they are actually used.

- The evaluated value is then memoized so that subsequent usages do not require re-evaluation.

Eager Evaluation

MinML is an eager language. Attempts to avoid “unnecessary” computation resulting from repeated evaluation of function arguments and data structure components, e.g., \( \lambda x : \text{nat}.x + x(x) \)

- Arguments to functions are evaluated before the function is evaluated.
- Components of data structures (in product types, sum types, etc.) must be values.
- Eager evaluation strategy is often referred to as call-by-value.

Language Options

- Lazy language: lazy evaluation of function application and data constructors.
- Eager language: eager evaluation of function application and data constructors.
- Mix-and-match: support various degrees of laziness in an eager language via recursive suspensions

Call-By-Need

Call-by-need is an evaluation strategy.

Can be applied to any language, e.g., MinML, with no new syntax or static semantics required.

Call-By-Need

Call-by-need is an optimized version of call-by-name:

- Bindings of all variables recorded in memory.
- When a binding is first needed, variable is evaluated and result re-bound to that variable.
- Subsequent uses of the variable just retrieve the stored value.

If a given binding is never needed, it is never evaluated, as befits call-by-name.
Dynamic Semantics for Call-By-Need

Abstract machine with states \((M, e)\), where

- \(M\) is a memory with domain \(\text{dom}(M)\).
- \(e\) is an open expression with free variables in \(\text{dom}(M)\).

Dynamic Semantics for Call-By-Need

- An \(e\) value
- \((M[x \mapsto e], x) \rightarrow (M[x \mapsto e], x)\)
- \((M[x \mapsto \bullet], e) \rightarrow (M'[x \mapsto \bullet], e')\)
- \((M[x \mapsto \bullet], x) \rightarrow (M'[x \mapsto \bullet], x)\)
- \((M, c_1) \rightarrow (M', c'_1)\)
- \((M, \text{app}(c_1, c_2)) \rightarrow (M', \text{app}(c'_1, c_2))\)

\(x \# M\)
\((M, \text{app}(\lambda(x \mapsto c), c_2)) \rightarrow (M[x \mapsto c_2], e)\)

Type Safety for Call-By-Need

Memory typing judgement:

- \(M : \Gamma\) iff \(M(x) = e\) implies \(x : \tau\) occurs in \(\Gamma\) for some \(\tau\) such that \(\Gamma \vdash e : \tau\)

Dynamic Semantics for Call-By-Need

- A memory is a finite function \(M : \text{Loc} \rightarrow \text{Expr}\) such that
  - the free variables in \(M(l)\) are in \(\text{dom}(M)\).

Initial state: \((\emptyset, e)\) where \(e\) is a closed expression

Final state: \((M, e)\) where \(e\) is an open value (variables are not values!)

Dynamic Semantics for Call-By-Need

- During evaluation of expression \(e\) bound to variable \(x\), binding of \(x\) is replaced by a "black hole"
  - Ensures that evaluation of \(e\) would get "stuck" if it required binding of \(x\).
  - Note that there are no transitions from \((M[x \mapsto \bullet], x)\)
  - Also ensures that any "fresh" variable added to memory during evaluation of \(e\) will be different from \(x\)

Type Safety for Call-By-Need

Well-formed machine state judgement:

- \((M, e)\) ok iff:
  1. There exists \(\Gamma\) and \(\tau\) such that \(M : \Gamma\) and \(\Gamma \vdash e : \tau\)
  2. If \(M(x) = e\), then \(x\) is not free in \(e\)
  3. If variable \(y\) is free in \(e\) or \(M(x)\) for some \(x\), then \(M(y) \neq \bullet\)

Unlike with reference cells, no self-reference allowed.
Safety for Call-By-Need

Theorem 1 (Preservation)
If \((M, e) \text{ ok}\) and \(M, e \rightarrow (M', e')\), then \((M', e') \text{ ok}\).

Proof is by induction on transition judgement.

Theorem 2 (Progress)
If \((M, e) \text{ ok}\) then either \((M, e) \text{ final}\) or there exists \((M', e')\) such that \((M, e) \rightarrow (M', e')\).

The proof is by induction on definition of \((M, e) \text{ ok}\).

General Recursion in Call-By-Need

Dynamic semantics in M Machine:

\[
\text{rec}(\tau, x. e) \rightarrow [x \rightarrow \text{rec}(\tau, x. e)] e
\]

Can use this to turn MinML recursive function into a derived form:

\[
\text{fun } f : (x : \tau_1) : \tau_2 \text{ is end}
\]

is taken to stand for

\[
\text{rec}(f : \tau_1 \rightarrow \tau_2, \lambda x : \tau_1. e)
\]

Yielding, as expected, the derived transition:

\[
\text{fun } f : (x : \tau_1) : \tau_2 \text{ is end}(e_1) \rightarrow^* [f, x \rightarrow \text{fun } f : (x : \tau_1) : \tau_2 \text{ is end}, e_1] e
\]

Safety for Call-By-Need General Recursion

Since \(x\) may occur free in \(e\), evaluation of \(e\) may require binding of \(x\), e.g., \(\text{rec}(x : \tau. x)\).

But this leads to stuck state \((M[x \rightarrow e], x)\), which is best viewed as a “checked error”.

Hence, only a weakened form of type safety can be proved for call-by-need semantics of MinML with general recursion. Since second and third well-formedness conditions for states cannot be maintained as invariant, evaluation may get stuck at a black hole. Progress therefore stated in terms of this “checked error” possibility.

General Recursion in Call-By-Need

MinML functions can call themselves, but more general recursion – self reference for arbitrary expressions – is possible with call-by-name since variables range over general expressions.

- Abstract syntax: \(\text{rec}(\tau, x. e)\)
- Concrete syntax: \(\text{rec}(x : \tau. e)\)
- Static semantics:

\[
\Gamma, x : \tau \vdash e : \tau
\]
Mix-and-Match Laziness

Call-by-need addresses lazy evaluation of function arguments.

Evaluation of data constructors (product, sum, recursive, etc.) can also be eager or lazy

- Independent of choice of call-by-value or call-by-need
- Independent of choice about other data constructors

Let’s make these choices on a per program rather than a per language basis, leaving the choices up to programmers!

Static Semantics for Suspension Types

\[\Gamma \vdash e : \tau\]

\[\Gamma \vdash \text{delay}(e) : \text{susp}(\tau)\]

\[\Gamma \vdash e : \text{susp}(\tau)\]

\[\Gamma \vdash \text{force}(e) : \tau\]

\[\Gamma, x : \text{susp}(\tau) \vdash e : \tau\]

\[\Gamma \vdash \text{rec}(x : \text{susp}(\tau).e) : \text{susp}(\tau)\]

Suspension Types

Segregate all aspects of laziness into a single type of memoized, suspended, self-referential computations.

Extend MinML expressions with this abstract syntax:

Types \(\tau ::= \text{susp}(\tau)\)

Exprs \(e ::= \text{delay}(e) | \text{force}(e) | \text{rec}(x : \text{susp}(\tau), e)\)

Note that general recursion is now limited to suspension types

Dynamic Semantics for Suspension Types

\[\langle M, \text{delay}(e) \rangle \mapsto \langle M[l \mapsto \text{delay}(e)], I \rangle\]

\[\langle M, e \rangle \mapsto \langle M', e' \rangle\]

\[\langle M, \text{force}(e) \rangle \mapsto \langle M', \text{force}(e') \rangle\]

\(\langle M[l \mapsto \text{delay}(e)], \text{force}(I) \rangle \mapsto \langle M'[l \mapsto \text{delay}(e'), \text{force}(I) \rangle\)

\[\langle M, \text{rec}(x : \text{susp}(\tau), e) \rangle \mapsto \langle M'[l \mapsto \text{delay}([x \leftarrow l]e)], I \rangle\]