Overview

A normal-order language

- Strictness
- Recursion
- Infinite data structures
- Direct denotational semantics
- Transition semantics
- Lazy (call-by-need) evaluation and its semantics
A Normal-Order Language

Recall that normal-order evaluation guarantees finding a canonical form if one exists; under eager evaluation more terms diverge.

So it is useful to consider a language based on normal-order evaluation.

Syntactically the language is similar to the eager functional language, but semantically

- all tuples and alternatives are canonical forms:

  - \( cfm ::= intcfm \mid boolcfm \mid funcfm \mid tupcfm \mid altcfm \)
  - \( tupcfm ::= \langle exp, \ldots exp \rangle \)  
    - **eager:** \( \langle cfm, \ldots cfm \rangle \)
  - \( altcfm ::= @ tag exp \)  
    - **eager:** \( @ tag cfm \)

  tuple and alternative constructors are lazy, evaluation is forced by deconstruction:

  \[
  e \Rightarrow \langle e_0, \ldots e_{n-1} \rangle \quad e_k \Rightarrow z \quad e . k \Rightarrow z \\
  \sumcase e \text{ of } (e_0, \ldots e_{n-1}) \Rightarrow z \quad (k < n)
  \]

- application is reduced in normal order:

  \[
  e \Rightarrow \lambda v. \tilde{e} \quad (\tilde{e}/v \rightarrow e' \Rightarrow z) \\
  e e' \Rightarrow z
  \]
Strictness of Constructs

A construct is strict in one of its subterm if its evaluation always requires the evaluation of that subterm since the meaning of a term created by this construct can be expressed as a strict function applied to the meaning of the subterm (recall a strict function between domains maps ⊥ to ⊥).

E.g. application is strict in the subterm in function position:

\[ e \Rightarrow \lambda v. \hat{e} \quad (\hat{e}/v \rightarrow e' \Rightarrow z) \]
\[ e \ e' \Rightarrow z \]
Semantics of Arithmetic and Boolean Operations

Transition semantics and strictness of arithmetic/Boolean constructs:

\[
\begin{align*}
e & \Rightarrow \lfloor n \rfloor \\
-e & \Rightarrow \lfloor -n \rfloor \\
e & \Rightarrow \lfloor n \rfloor \quad e' & \Rightarrow \lfloor n' \rfloor \\
e + e' & \Rightarrow \lfloor n + n' \rfloor \\
e & \Rightarrow \text{true} \quad e' & \Rightarrow z \\
\text{if } e \text{ then } e' \text{ else } e'' & \Rightarrow z \\
e & \Rightarrow \text{false} \quad e'' & \Rightarrow z \\
\text{if } e \text{ then } e' \text{ else } e'' & \Rightarrow z
\end{align*}
\]
Recursion in the Normal-Order Language

Additional syntax:

\[
exp ::= \ldots | \text{rec } exp
\]

with evaluation rule

\[
\frac{e (\text{rec } e) \Rightarrow z}{\text{rec } e \Rightarrow z}
\]

Thus \text{rec } e is syntactic sugar for \( Y e \) in untyped languages or those with recursive types, but must be primitive in a simply typed language.

More syntactic sugar:

\[
\text{letrec } p_1 \equiv e_1, \ldots p_n \equiv e_n \text{ in } e
\]

\[
\text{def } \equiv \text{let } \langle p_1, \ldots p_n \rangle \equiv \text{rec } (\lambda \langle p_1, \ldots p_n \rangle. \langle e_1, \ldots e_n \rangle) \text{ in } e
\]

Note: fixed-points are not necessarily functions since all tuples and alternatives are canonical forms, so e.g. the following does not diverge:

\[
\text{letrec ones } \equiv 1 : : \text{ones in ones}
\]
Using Normal-Order Evaluation

One can define short-circuit Boolean operators as syntactic sugar:

\[ \neg e \overset{\text{def}}{=} \text{if } e \text{ then false else true} \]

\[ e \land e' \overset{\text{def}}{=} \text{if } e \text{ then } e' \text{ else false} \]

Since only the subterms whose value is necessary are evaluated, we can use the standard \( \text{foldl} \) (reduce) to define short-circuit operations on lists:

\[
\text{let } \text{foldl} \equiv \lambda f. \lambda z. \text{rec}(\lambda g. \lambda l. \text{listcase } l \text{ of } (z. \lambda x. \lambda x s. f x (g z x s)))),
\]

\[
\text{prod} \equiv \text{foldl}(\lambda x. \lambda y. \text{if } x = 0 \text{ then } 0 \text{ else } x * y) \ 1
\]

\text{in } \ldots
Since fixed points can be of any type, one can define

\[
\text{letrec}
\]
\[
\begin{align*}
nats &\equiv 0 :: \text{map} (\lambda x. x+1) \ nats, \\
sumlists &\equiv \lambda xs. \lambda ys. \text{listcase} \ xs \ of \\
&\quad (ys, \\
&\quad \quad \lambda x. \lambda xs'. \text{listcase} \ ys \ of \ (\text{nil}, \lambda y. \lambda ys'. \ (x+y) :: \text{sumlist} \ xs \ ys'))), \\
fib &\equiv 0 :: fib1, \\
fib1 &\equiv 1 :: \text{sumlists} \ fib \ fib1
\end{align*}
\]

in ... 

Any finite part of these data structures will be computed in finite time
as the canonical forms of its subterms become necessary.
Direct Denotational Semantics

The major change from the semantics of an eager language is that the environments bind variables to computations in $V_*$, not values in $V$:

$$E \overset{\text{def}}{=} \text{var} \rightarrow V_*$$

Similarly the (pre)domains of functions, tuples, and alternatives are

- $V_{\text{fun}} = [V_* \rightarrow V_*]$ since variables are bound to computations
- $V_{\text{tup}} = (V_*)^*$ since tuples...
- $V_{\text{alt}} = \mathbb{N} \times V_*$ and alternatives are non-strict

Semantic equations for abstraction and application:

$$\llbracket \lambda v. \ e \rrbracket_\eta = \iota_{\text{norm}} (\iota_{\text{fun}} (\lambda a \in V_. \llbracket e \rrbracket_\eta | v : a))$$

$$\llbracket e \ e' \rrbracket_\eta = (\lambda f \in V_{\text{fun}}. f (\llbracket e' \rrbracket_\eta))_{\text{fun}*} (\llbracket e \rrbracket_\eta)$$
Semantics of Tuples and Alternatives

Unlike the eager language, tuples and alternatives are non-strict, hence divergence in subterms of tuples and alternatives is not propagated to the meaning of the entire term:

\[
\begin{align*}
\llbracket \langle e_0, \ldots, e_{n-1} \rangle \rrbracket_\eta &= \iota_{\text{norm}}(\iota_{\text{tup}}(\llbracket e_0 \rrbracket_\eta, \ldots, \llbracket e_{n-1} \rrbracket_\eta)) \\
\llbracket @ k e \rrbracket_\eta &= \iota_{\text{norm}}(\iota_{\text{alt}}(k, \llbracket e \rrbracket_\eta))
\end{align*}
\]

The meaning of selecting a component of a tuple is simply the meaning of the component, so no extra care is needed to propagate its divergence:

\[
\llbracket e.k \rrbracket_\eta = (\lambda t \in V_{\text{tup}}. \text{if } k \in \text{dom } t \text{ then } t_k \text{ else tyerr})_{\text{tup}*}(\llbracket e \rrbracket_\eta)
\]

The sumcase construct is strict in the discriminant, hence the use of lifting:

\[
\begin{align*}
\llbracket \text{sumcase } e \text{ of } (e_0, \ldots, e_{n-1}) \rrbracket_\eta \\
= (\lambda \langle k, a \rangle \in V_{\text{alt}}. \text{if } k < n \text{ then } (\lambda f \in V_{\text{fun}}. f a)_{\text{fun}*}(\llbracket e_k \rrbracket_\eta) \text{ else tyerr})_{\text{alt}*}(\llbracket e \rrbracket_\eta)
\end{align*}
\]
The normal-order evaluation of a term can be described by a transition relation $\rightarrow$:

\[
(\lambda v. e) e' \rightarrow (e/v \rightarrow e')
\]

\[
\text{if true then } e \text{ else } e' \rightarrow e
\]

\[
\text{if false then } e \text{ else } e' \rightarrow e'
\]

\[
\langle e_0, \ldots e_{n-1} \rangle.k \rightarrow e_k, \quad \text{if } k < n
\]

\[
\text{sumcase } \mathbin{\@} k e \text{ of } (e_0, \ldots e_{n-1}) \rightarrow e_k e, \quad \text{if } k < n
\]

\[
\text{rec } e \rightarrow e (\text{rec } e)
\]

plus contextual closure according to the strictness of constructs, e.g.

\[
e \rightarrow e_1
\]

\[
e e' \rightarrow e_1 e'
\]

\[
e \Rightarrow e_1
\]

\[
\text{if } e \text{ then } e' \text{ else } e'' \Rightarrow \text{if } e_1 \text{ then } e' \text{ else } e''
\]
Lazy (Call-By-Need) Evaluation

A naïve substitution-based implementation of normal-order evaluation would be very inefficient because it would duplicate much of the work:

\[
\begin{align*}
e \Rightarrow [n] \\
(\lambda x. x+x) e \Rightarrow [n + n]
\end{align*}
\]

will evaluate \(e\) to \([n]\) twice;

in general the extra amount of work is not bounded by any elementary function. Call-by-need evaluation uses sharing to reduce each term at most once:

Hence we need to extend the language so it can express sharing.
The Call-By-Need Calculus

For simplicity the syntax of application and alternatives is restricted to

\[ exp ::= \ldots \mid \text{exp \ var} \mid \text{@ \ tag \ var} \]

but letrec is promoted to a basic construct,
so we can define the general forms of application and alternatives as sugar:

\[\begin{align*}
\text{letrec } v & \equiv e' \ \text{in } e v \\
\text{letrec } v & \equiv e \ \text{in } \text{@ } k v \\
\text{letrec } v & \equiv e v \ \text{in } v
\end{align*}\]

where \( v \notin FV(e) \).

We also rename as necessary so that no bindings in a term bind the same variables.
Semantics of the Call-By-Need Calculus

Sharing is expressed by referring to variables, which are mapped to terms by heaps:

$$\sigma \in \text{var} \rightarrow (\text{exp} \cup \{\text{busy}\})$$

A heap $\sigma$ is closed if $FV(\sigma v) \subseteq \text{dom } \sigma$ for each $v \in \text{dom } \sigma$.

A term $e$ and a heap $\sigma$ are compatible if $\sigma$ is closed and $FV(e) \subseteq \text{dom } \sigma$.

Evaluation of a variable $v$ re-maps it to the value of the term that $v$ is bound to:

$$\left< [\sigma | v : \text{busy}], e \right> \Rightarrow \left< z, \sigma' \right>$$

$$\left< [\sigma | v : e], v \right> \Rightarrow \left< z, [\sigma' | v : z_{\text{renamed}}] \right>$$

Implementations usually avoid reevaluating values by marking them as such. The token busy is used to prevent attempts to compute the value of a variable whose value is currently being computed (busy is not really necessary since we don’t allow infinite derivations anyway).

Evaluation of letrec moves the bindings to the heap:

$$\left< [\sigma | v_0 : e_0 | \ldots], e \right> \Rightarrow \left< z, \sigma' \right>$$

$$\left< \sigma, \text{letrec } v_0 \equiv e_0, \ldots \text{ in } e \right> \Rightarrow \left< z, \sigma' \right>$$
A Call-By-Need Example

\((\lambda f\ 4+f\ 2)\ ((\lambda x.\ \lambda y.\ (x+2)*y)\ 5)\)

expands as \(\text{letrec } g \equiv (\text{letrec } z \equiv 5 \text{ in } (\lambda x.\ \lambda y.\ (x+2)*y)\ z)\ \text{in } (\lambda f\ 4+f\ 2)\ g\)

\[
\langle [], \text{letrec } g \equiv (\text{letrec } z \equiv 5 \text{ in } (\lambda x.\ \lambda y.\ (x+2)*y)\ z)\ \text{in } (\lambda f\ 4+f\ 2)\ g\rangle
\]

\[
\langle [g : \text{letrec } z \equiv 5 \text{ in } (\lambda x.\ \lambda y.\ (x+2)*y)\ z], \ (\lambda f\ 4+f\ 2)\ g\rangle
\]

\[
\Rightarrow \langle \lambda f\ 4+f\ 2, \ [g : \text{letrec } z \equiv 5 \text{ in } (\lambda x.\ \lambda y.\ (x+2)*y)\ z]\rangle
\]

\[
\langle [g : \text{letrec } z \equiv 5 \text{ in } (\lambda x.\ \lambda y.\ (x+2)*y)\ z], \ g \ 4+g\ 2\rangle
\]

\[
\Rightarrow \langle \lambda y.\ (z+2)*y, \ [g : \lambda y.\ (z+2)*y \ | \ z : 5]\rangle
\]

\[
\langle [g : \lambda y.\ (z+2)*y \ | \ z : 5], \ (\lambda y.\ (z+2)*y)\ 4\rangle
\]

\[
\langle [g : \lambda y.\ (z+2)*y \ | \ z : 5], \ (z+2)*4\rangle
\]

\[
\Rightarrow \langle 28, \ [g : \lambda y.\ (z+2)*y \ | \ z : 5]\rangle
\]

\[
\langle [g : \lambda y.\ (z+2)*y \ | \ z : 5], \ g \ 2\rangle
\]

\[
\Rightarrow \langle \lambda y.\ (z+2)*y, \ [g : \lambda y.\ (z+2)*y \ | \ z : 5]\rangle
\]

\[
\Rightarrow \langle 14, \ [g : \lambda y.\ (z+2)*y \ | \ z : 5]\rangle
\]

\[
\Rightarrow \langle 42, \ [g : \lambda y.\ (z+2)*y \ | \ z : 5]\rangle
\]

— evaluation of \(g\) begins

— evaluation of \(g\) ends

— \(g\) is already evaluated
A Remark on Lazy Evaluation

Note that in the example the term \((z+2)\) is evaluated twice, although the heap binding of \(z\) to 5 is unchanged, so the work for evaluating \((z+2)\) is duplicated.

Call-by-need cannot completely eliminate this kind of duplication of work.

This can be achieved using optimal evaluation [Lévy 78, Lamping 90, Gonthier/Abadi/Lévy 92].