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:

$$\begin{array}{rcll} cfm & ::= & intcfm \mid boolcfm \mid funcfm \mid tupcfm \mid altcfm & \\ tupcfm & ::= & \langle exp, \ldots\ exp \rangle & \text{eager: } \langle cfm, \ldots\ cfm \rangle \\ altcfm & ::= & \mathbf{@}\ tag\ exp & \text{eager: } \mathbf{@}\ tag\ cfm \end{array}$$

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

$$\frac{e \Rightarrow \langle e_0, \ldots\ e_{n-1} \rangle \quad e_k \Rightarrow z}{e\,.\,k \Rightarrow z} \qquad \frac{e \Rightarrow \mathbf{@}\ k\ e' \quad e_k\, e' \Rightarrow z}{\mathbf{sumcase}\ e\ \mathbf{of}\ (e_0, \ldots\ e_{n-1}) \Rightarrow z} \qquad (k < n)$$

- application is reduced in normal order: $$\frac{e \Rightarrow \lambda v.\, \hat{e} \quad (\hat{e}/v \rightarrow e' \Rightarrow z)}{e\, e' \Rightarrow z}$$

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 $\perp$ to $\perp$).

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

$$\frac{e \Rightarrow \lambda v.\, \hat{e} \quad (\hat{e}/v \rightarrow e' \Rightarrow z)}{e\, e' \Rightarrow z}$$

Semantics of Arithmetic and Boolean Operations

Transition semantics and strictness of arithmetic/Boolean constructs:

$$\frac{e \Rightarrow \lfloor n \rfloor}{\texttt{-}e \Rightarrow \lfloor -n \rfloor}$$

$$\frac{e \Rightarrow \lfloor n \rfloor \quad e' \Rightarrow \lfloor n' \rfloor}{e\texttt{+}e' \Rightarrow \lfloor n + n' \rfloor}$$

$$\frac{e \Rightarrow \mathbf{true} \quad e' \Rightarrow z}{\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' \Rightarrow z}$$

$$\frac{e \Rightarrow \mathbf{false} \quad e'' \Rightarrow z}{\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' \Rightarrow z}$$

Recursion in the Normal-Order Language

Additional syntax:

$$exp ::= \ldots \mid \mathbf{rec}\, exp$$

with evaluation rule

$$\frac{e\,(\mathbf{rec}\, e) \Rightarrow z}{\mathbf{rec}\, e \Rightarrow z}$$

Thus $\mathbf{rec}\, e$ 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:

$$\mathbf{letrec}\ p_1 \equiv e_1,\ \ldots\ p_n \equiv e_n\ \mathbf{in}\ e$$

$$\stackrel{\text{def}}{=}\quad \mathbf{let}\ \langle p_1,\ \ldots\ p_n\rangle \equiv \mathbf{rec}\,(\lambda\langle p_1,\ \ldots\ p_n\rangle.\ \langle e_1,\ \ldots\ e_n\rangle)\ \mathbf{in}\ e$$

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

$$\mathbf{letrec}\ \mathsf{ones} \equiv \mathsf{1}::\mathsf{ones}\ \mathbf{in}\ \mathsf{ones}$$

Using Normal-Order Evaluation

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

$$\neg\, e \stackrel{\text{def}}{=} \textbf{if } e \textbf{ then false else true}$$

$$e \wedge e' \stackrel{\text{def}}{=} \textbf{if } e \textbf{ then } e' \textbf{ else false}$$

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

$$\begin{array}{l} \textbf{let } \mathsf{foldl} \equiv \lambda \mathsf{f}.\, \lambda \mathsf{z}.\, \textbf{rec}\, (\lambda \mathsf{g}.\, \lambda \mathsf{l}.\, \textbf{listcase}\ \mathsf{l}\ \textbf{of}\ (\mathsf{z},\ \lambda \mathsf{x}.\, \lambda \mathsf{xs}.\, \mathsf{f}\,\mathsf{x}\,(\mathsf{g}\,\mathsf{z}\,\mathsf{xs}))), \\ \quad\ \mathsf{prod} \equiv \mathsf{foldl}\, (\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \textbf{if}\ \mathsf{x} = 0\ \textbf{then}\ 0\ \textbf{else}\ \mathsf{x} {*} \mathsf{y})\ 1 \\ \textbf{in}\ \ldots \end{array}$$

Programming with Infinite Data Structures

Since fixed points can be of any type, one can define

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{letrec}} \\
\quad \mathsf{nats} & \equiv & \mathsf{0} :: \mathsf{map}\,(\lambda \mathsf{x}.\, \mathsf{x{+}1})\,\mathsf{nats}, \\
\quad \mathsf{sumlists} & \equiv & \lambda \mathsf{xs}.\, \lambda \mathsf{ys}.\, \mathbf{listcase}\ \mathsf{xs}\ \mathbf{of} \\
 & & \qquad\qquad (\mathsf{ys}, \\
 & & \qquad\qquad\ \lambda \mathsf{x}.\, \lambda \mathsf{xs}'.\, \mathbf{listcase}\ \mathsf{ys}\ \mathbf{of}\ (\mathbf{nil},\ \lambda \mathsf{y}.\, \lambda \mathsf{ys}'.\, (\mathsf{x{+}y}) :: \mathsf{sumlist}\ \mathsf{xs}\ \mathsf{ys}')), \\
\quad \mathsf{fib} & \equiv & \mathsf{0} :: \mathsf{fib1}, \\
\quad \mathsf{fib1} & \equiv & \mathsf{1} :: \mathsf{sumlists}\ \mathsf{fib}\ \mathsf{fib1} \\
\multicolumn{3}{l}{\ \mathbf{in}\ \ldots}
\end{array}
$$

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:

$$E \stackrel{\text{def}}{=} var \to V_*$$

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

$$\begin{array}{rcll} V_{fun} & = & [V_* \to V_*] & \text{since variables are bound to computations} \\ V_{tup} & = & (V_*)^* & \text{since tuples\ldots} \\ V_{alt} & = & \mathbf{N} \times V_* & \text{and alternatives are non-strict} \end{array}$$

Semantic equations for abstraction and application:

$$\begin{array}{rcl} [\![\lambda v.\, e]\!]\eta & = & \iota_{norm}\,(\iota_{fun}\,(\lambda a \in V_*.\, [\![e]\!][\eta \mid v : a])) \\ [\![e\, e']\!]\eta & = & (\lambda f \in V_{fun}.\, f\,([\![e']\!]\eta))_{fun*}\,([\![e]\!]\eta) \end{array}$$

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:

$$[\![\langle e_0, \ldots e_{n-1}\rangle]\!]\eta \;=\; \iota_{norm}\,(\iota_{tup}\,\langle [\![e_0]\!]\eta, \ldots [\![e_{n-1}]\!]\eta\rangle)$$

$$[\![@\;k\;e]\!]\eta \;=\; \iota_{norm}\,(\iota_{alt}\,\langle k, [\![e]\!]\eta\rangle)$$

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:

$$[\![e\,.\,k]\!]\eta \;=\; (\lambda t \in V_{tup}.\,\mathbf{if}\ k \in \mathrm{dom}\ t\ \mathbf{then}\ t\,k\ \mathbf{else}\ \mathrm{tyerr})_{tuple*}\,([\![e]\!]\eta)$$

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

$$[\![\mathbf{sumcase}\ e\ \mathbf{of}\ (e_0, \ldots e_{n-1})]\!]\eta$$
$$=\; (\lambda\langle k, a\rangle \in V_{alt}.\,\mathbf{if}\ k < n\ \mathbf{then}\ (\lambda f \in V_{fun}.\, f\,a)_{fun*}\,([\![e_k]\!]\eta)\ \mathbf{else}\ \mathrm{tyerr})_{alt*}\,([\![e]\!]\eta)$$

Transition Semantics

The normal-order evaluation of a term can be described by a transition relation $\mapsto$:

$$
\begin{array}{rcll}
(\lambda v.\, e)\, e' & \mapsto & (e/v \to e') & \\
\mathbf{if\ true\ then}\ e\ \mathbf{else}\ e' & \mapsto & e & \\
\mathbf{if\ false\ then}\ e\ \mathbf{else}\ e' & \mapsto & e' & \\
\langle e_0,\ \ldots\ e_{n-1}\rangle\,.\,k & \mapsto & e_k, & \text{if}\ \ k < n \\
\mathbf{sumcase}\ @\ k\ e\ \mathbf{of}\ (e_0,\ \ldots e_{n-1}) & \mapsto & e_k\, e, & \text{if}\ \ k < n \\
\mathbf{rec}\, e & \mapsto & e\, (\mathbf{rec}\, e) &
\end{array}
$$

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

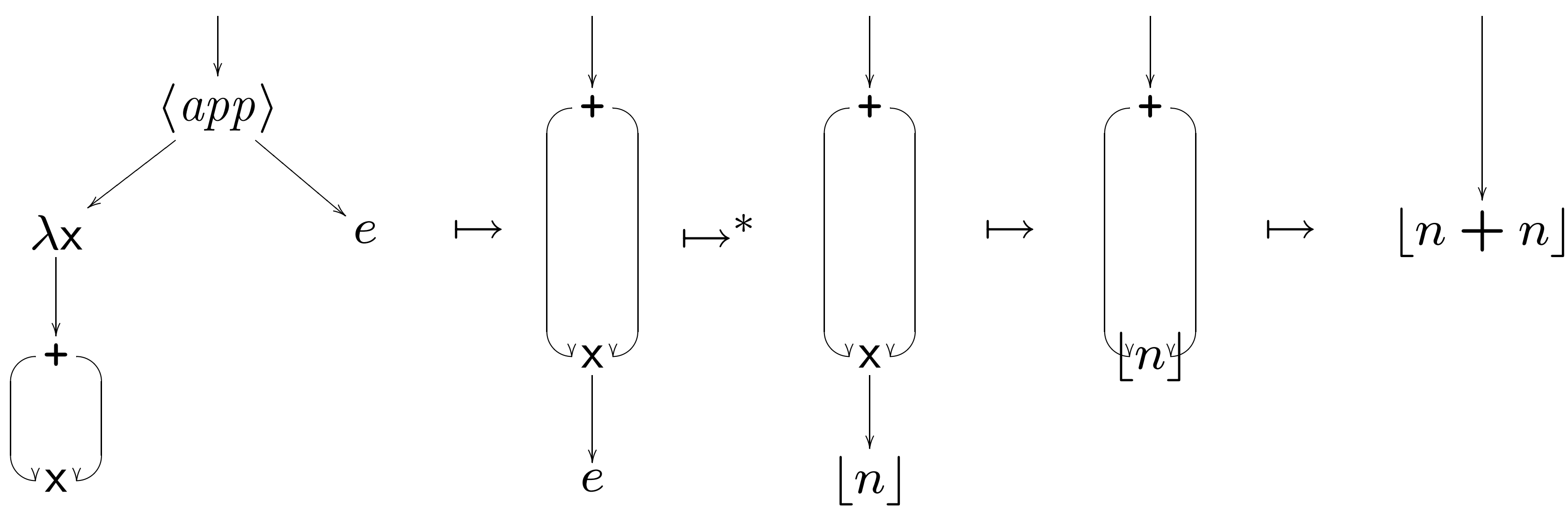
$$
\frac{e \mapsto e_1}{e\, e' \mapsto e_1\, e'} \qquad \frac{e \Rightarrow e_1}{\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' \Rightarrow \mathbf{if}\ e_1\ \mathbf{then}\ e'\ \mathbf{else}\ e''}
$$

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:

$$\frac{e \Rightarrow \lfloor n \rfloor}{(\lambda \mathsf{x}.\, \mathsf{x}{+}\mathsf{x})\, e \Rightarrow \lfloor n + n \rfloor}$$

will evaluate e to $\lfloor n \rfloor$ 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

$$exp ::= \ldots \mid exp\ var \mid @\ tag\ var$$

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

$$\begin{aligned} e\, e' &\stackrel{\text{def}}{=} \textbf{letrec}\ v \equiv e'\ \textbf{in}\ e\, v \\ @\ k\ e &\stackrel{\text{def}}{=} \textbf{letrec}\ v \equiv e\ \textbf{in}\ @\ k\ v \\ \textbf{rec}\, e &\stackrel{\text{def}}{=} \textbf{letrec}\ v \equiv e\, v\ \textbf{in}\ v \end{aligned}$$

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:

$$\sigma \in \mathit{var} \to (\mathit{exp} \cup \{\mathbf{busy}\})$$

A heap σ is closed if $FV(\sigma\, v) \subseteq \mathrm{dom}\ \sigma$ for each $v \in \mathrm{dom}\ \sigma$.
A term e and a heap σ are compatible if σ is closed and $FV(e) \subseteq \mathrm{dom}\ \sigma$.

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

$$\frac{\langle [\sigma \mid v : \mathbf{busy}],\ e\rangle \Rightarrow \langle z,\ \sigma'\rangle}{\langle [\sigma \mid v : e],\ v\rangle \Rightarrow \langle z,\ [\sigma' \mid v : z_{renamed}]\rangle}$$

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:

$$\frac{\langle [\sigma | v_0 : e_0 | \ldots],\ e\rangle \Rightarrow \langle z, \sigma'\rangle}{\langle \sigma,\ \mathbf{letrec}\ v_0 \equiv e_0, \ldots\ \mathbf{in}\ e\rangle \Rightarrow \langle z, \sigma'\rangle}$$

A Call-By-Need Example

$(\lambda \mathsf{f}.\, \mathsf{f}\ 4{+}\mathsf{f}\ 2)\ ((\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, (\mathsf{x}{+}2){*}\mathsf{y})\ 5)$

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

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

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

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

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

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

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

$\quad\quad\quad\quad \langle [\mathsf{g} : \textbf{letrec}\ \mathsf{z} \equiv 5\ \textbf{in}\ (\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, (\mathsf{x}{+}2){*}\mathsf{y})\ \mathsf{z}],\ \mathsf{g} \rangle$ — evaluation of g begins

$\quad\quad\quad\quad\quad \langle [\mathsf{g} : \textbf{busy}],\ \textbf{letrec}\ \mathsf{z} \equiv 5\ \textbf{in}\ (\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, (\mathsf{x}{+}2){*}\mathsf{y})\ \mathsf{z} \rangle$

$\quad\quad\quad\quad\quad\quad \langle [\mathsf{g} : \textbf{busy} \mid \mathsf{z} : 5],\ (\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, (\mathsf{x}{+}2){*}\mathsf{y})\ \mathsf{z} \rangle$

$\quad\quad\quad\quad\quad\quad\quad \langle [\mathsf{g} : \textbf{busy} \mid \mathsf{z} : 5],\ \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \rangle$

$\quad\quad\quad\quad \Rightarrow \langle \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y},\ [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5] \rangle$ — evaluation of g ends

$\quad\quad\quad\quad \langle [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5],\ (\lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y})\ 4 \rangle$

$\quad\quad\quad\quad\quad \langle [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5],\ (\mathsf{z}{+}2){*}4 \rangle$

$\quad\quad\quad\quad\quad \ldots$

$\quad\quad\quad \Rightarrow \langle 28,\ [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5] \rangle$

$\quad\quad\quad \langle [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5],\ \mathsf{g}\ 2 \rangle$

$\quad\quad\quad\quad \langle [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5],\ \mathsf{g} \rangle$ — g is already evaluated

$\quad\quad\quad\quad \Rightarrow \langle \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y},\ [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5] \rangle$

$\quad\quad\quad\quad \ldots$

$\quad\quad\quad \Rightarrow \langle 14,\ [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5] \rangle$

$\Rightarrow \langle 42,\ [\mathsf{g} : \lambda \mathsf{y}.\, (\mathsf{z}{+}2){*}\mathsf{y} \mid \mathsf{z} : 5] \rangle$

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].