Reduction Strategies in the Lambda Calculus

A Systematic Approach to Their Specification and Efficient Implementation with Abstract Machines

Małgorzata Biernacka

Institute of Computer Science, University of Wrocław

WEPN 2023

Reduction strategies

- one-step strategy is a function $F : \Lambda \to \Lambda$ s.t.

$$t \to_\beta^1 F(t)$$

- examples: leftmost-outermost, call by name, call by value

Reduction strategies

- of interest both in theory and in practice of lambda calculus
- multitude of strategies, defined and studied in different disguises
- relevant for efficient computation in lambda calculus
- often used as folklore, auxiliary tool

Reduction strategies – systematic approach

- formats to define strategies
- methodology to interderive various semantics (based on functional programming)
- formalization and classification

Weak strategies

$$\underline{(\lambda xy.xy)((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{CbN}} \lambda y.(\lambda z.z)(\lambda w.w)y$$

Weak strategies

$$\underline{(\lambda xy.xy)((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{CbN}} \lambda y.(\lambda z.z)(\lambda w.w)y$$

$$(\lambda xy.xy)\underline{((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{CbV}} \underline{(\lambda xy.xy)\,(\lambda w.w)}$$

Weak strategies

$$\underline{(\lambda xy.xy)((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{CbN}} \lambda y.(\lambda z.z)(\lambda w.w)y$$

$$(\lambda xy.xy)\underline{((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{CbV}} \underline{(\lambda xy.xy)\,(\lambda w.w)} \xrightarrow{\text{CbV}} (\lambda y.(\lambda w.w)y)$$

Strong strategies

- fully normalize terms
- need to descend under λ and account for free variables in terms
- conservative extensions of weak strategies (e.g., strong CbN, strong CbV, strong CbNeed)
- efficient implementations required e.g. for typechecking in dependent types

Strong strategies

Normal order = "iterate CbN"

$$\underline{(\lambda xy.xy)\color{blue}{((\lambda z.z)(\lambda w.w))}} \xrightarrow{\text{NO}} \lambda y.\underline{\color{blue}{(\lambda z.z)(\lambda w.w)}}y$$

Strong strategies

Normal order = "iterate CbN"

$$\underline{(\lambda xy.xy)((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{NO}} \lambda y.\underline{(\lambda z.z)(\lambda w.w)}y \xrightarrow{\text{NO}} \lambda y.\underline{(\lambda w.w)y} \xrightarrow{\text{NO}} \lambda y.y$$

Strong strategies

$$(\lambda xy.x(yy))\underline{((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{SCbV}} \underline{(\lambda xy.x(yy))\,(\lambda w.w)}$$

Strong strategies

$$(\lambda xy.x(yy))\underline{((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{SCbV}} \underline{(\lambda xy.x(yy))\,(\lambda w.w)}$$
$$\xrightarrow{\text{SCbV}} \lambda y.\underline{(\lambda w.w)(yy)}$$

Strong strategies

$$(\lambda xy.x(yy))\underline{((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{SCbV}} \underline{(\lambda xy.x(yy))\,(\lambda w.w)}$$

$$\xrightarrow{\text{SCbV}} \lambda y.\underline{(\lambda w.w)(yy)}$$

$$\xrightarrow{\text{SCbV}} \lambda y.yy$$

Strong strategies

$$(\lambda xy.x(yy))\underline{((\lambda z.z)(\lambda w.w))} \xrightarrow{\text{SCbV}} \underline{(\lambda xy.x(yy))\,(\lambda w.w)}$$
$$\xrightarrow{\text{SCbV}} \lambda y.\underline{(\lambda w.w)(yy)}$$
$$\xrightarrow{\text{SCbV}} \lambda y.yy$$

Strong CbV = iterate Open CbV

Strong strategies

$$
\begin{aligned}
(\lambda xy.x(yy))\underline{((\lambda z.z)(\lambda w.w))} &\xrightarrow{\text{SCbV}} \underline{(\lambda xy.x(yy))\,(\lambda w.w)} \\
&\xrightarrow{\text{SCbV}} \lambda y.\underline{(\lambda w.w)(yy)} \\
&\xrightarrow{\text{SCbV}} \lambda y.yy
\end{aligned}
$$

Strong CbV = iterate Open CbV

Strong CbV is nondeterministic – we can choose right-to-left normalization (rrSCbV)

Formats of operational semantics

- structural operational semantics
- big-step semantics
- reduction semantics
- abstract machine
- definitional interpreter

Big-step semantics

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t_1 \Downarrow \lambda x.t \qquad t_2 \Downarrow t_2' \qquad t[x := t_2'] \Downarrow t'}{t_1\ t_2 \Downarrow t'}$$

$$\frac{}{x \Downarrow x} \qquad \frac{t_1 \Downarrow t_1' \not\equiv \lambda x.t \qquad t_2 \Downarrow t_2'}{t_1\ t_2 \Downarrow t_1'\ t_2'}$$

Open call by value

Reduction semantics

$$w ::= \lambda x.t \mid x\,\vec{w} \qquad E ::= \Box \;\mid\; w\,E \;\mid\; E\,t$$

$$\frac{}{(\lambda x.t)\,w \rightharpoonup_{\beta_w} t[x := w]} \qquad \frac{t \rightharpoonup_{\beta_w} t'}{E[t] \overset{lcbw}{\to} E[t']}$$

Left-to-right open call by value

Abstract machines

- micro-step semantics (explicit decomposition and substitution)
- abstract model of language implementation
- work on source terms (not on compiled terms)
- constant cost of each transition
- abstract cost model of computation

Krivine machine for CbN evaluation

$$t ::= n \mid t\,t \mid \lambda t \qquad C ::= [t, E]$$

$$E ::= \bullet \mid C :: E$$
$$S ::= \bullet \mid C :: S$$

$$\begin{aligned}
t &\mapsto \langle t, \bullet, \bullet \rangle \\
\langle t_1\, t_2, E, S \rangle &\to \langle t_1, E, [t_2, E] :: S \rangle \\
\langle \lambda t, E, C :: S \rangle &\to \langle t, C :: E, S \rangle \\
\langle 0, [t, E] :: E', S \rangle &\to \langle t, E, S \rangle \\
\langle n + 1, C :: E, S \rangle &\to \langle n, E, S \rangle
\end{aligned}$$

Techniques for AM derivation

- refocusing – from reduction semantics to abstract machine
- functional correspondence – from higher-order normalizer to abstract machine
- introduced for weak strategies, extendable to strong ones

Generalized reduction semantics

Normal order in λ-calculus

$$t ::= x \mid \lambda x.\, t \mid t\, t \qquad a ::= x \mid a\, n \qquad n ::= a \mid \lambda x.\, n$$

$$\begin{array}{rcl} \underline{E} & ::= & F \mid \lambda x.E \mid a\, E \\ F & ::= & \square_F \mid F\, t \end{array}$$

$$E[(\lambda x.t)\, s] \;\stackrel{no}{\to}\; E[t[x := s]]$$

Generalized reduction semantics – formalization

syntactic categories: **kinds**, initial kind, terms

values, potential redices,

elementary contexts – parameterized by kinds

atomic plug – defining meaning of contexts

contraction function

proofs of basic properties

```
Parameters (ckind term : Set)
      (init_ckind : ckind)
      (redex value : ckind -> Set).
Parameters
      (elem_ctx : ckind -> ckind -> Set)
      (elem_plug : ∀ {k0 k1}, term ->
       elem_ctx k0 k1 -> term).

Parameter contract :
       ∀ {k}, redex k -> option term.

Axioms
(v_triv: ∀ ..., ec:[t] = v -> ∃ v', t = v')
(v_red: ∀ {k} (v : value k) (r : redex k),
v <> r).
```

Input to generalized refocusing

- generalized reduction semantics
- linear strict order $<_{k,t}$ on instances of productions from P^k that are compatible with t (i.e., elementary k-contexts matching t)
- atomic decomposition functions
- conditions on input enforce unique decomposition

Input to generalized refocusing — normal order

- elementary contexts

$$E ::= \lambda x.\square_E \mid a\ \square_E \mid \square_F\ t$$

$$F ::= \square_F\ t$$

- search order $a\ \square_E <_{E,\_}\ \square_F\ t$

Abstract machine for normal order

$$
\begin{aligned}
\langle \lambda x.t, C, F \rangle_{\mathsf{e}} &\triangleright \langle C, F, \lambda x.t \rangle_{\mathsf{c}} \\
\langle \lambda x.t, C, E \rangle_{\mathsf{e}} &\triangleright \langle t, \lambda x.\Box :: C, E \rangle_{\mathsf{e}} \\
\langle t_1\, t_2, C, k \rangle_{\mathsf{e}} &\triangleright \langle t_1, (k, \Box\, t_2) :: C, F \rangle_{\mathsf{e}} \\
\langle x, C, k \rangle_{\mathsf{e}} &\triangleright \langle C, k, x \rangle_{\mathsf{c}} \\
\langle \lambda x.\Box :: C, E, v \rangle_{\mathsf{c}} &\triangleright \langle C, E, \lambda x.v \rangle_{\mathsf{c}} \\
\langle n_e\, \Box :: C, E, v \rangle_{\mathsf{c}} &\triangleright \langle C, E, n_e\, v \rangle_{\mathsf{c}} \\
\langle (k, \Box\, s) :: C, F, \lambda x.t \rangle_{\mathsf{c}} &\triangleright \langle t[x := s], C, k \rangle_{\mathsf{e}} \\
\langle (k, \Box\, s) :: C, k', x \rangle_{\mathsf{c}} &\triangleright \langle s, (k, x\, \Box) :: C, E \rangle_{\mathsf{e}}
\end{aligned}
$$

Correctness – intensionally

An abstract rewriting system $\langle \mathcal{S}, \Rightarrow \rangle$ **traces** another system $\langle \mathcal{T}, \to \rangle$ if there exists a surjection $[\![]\!] : \mathcal{S} \to \mathcal{T}$ s.t.

1. if $s_1 \Rightarrow s_2$ then $[\![s_1]\!] = [\![s_2]\!]$ or $[\![s_1]\!] \to [\![s_2]\!]$
2. if $t_1 \to t_2$ then for each s_0 s.t. $[\![s_0]\!] = t_1$ there exists $s_0 \Rightarrow \ldots \Rightarrow s_{n+1}$, where $[\![s_0]\!] = \ldots = [\![s_n]\!]$ and $[\![s_{n+1}]\!] = t_2$
3. there are no silent loops

Theorem

Let M – machine generated by generalized refocusing from a RS with terms $\mathcal{T}$ and reduction relation $\to$. Then M traces $\langle \mathcal{T}, \to \rangle$.

Functional correspondence

Higher-order evaluator

Closure conversion

CPS translation

Defunctionalization

↓

Abstract machine

A call-by-name evaluator

```
type value = Value of (thunk -> value)
 and thunk = unit -> value

let rec eval (e : env) (t : term) : value =
  match t with
  | Var n        -> List.nth e n ()
  | App(t0, t1) -> value_unfold (eval e t0)
                                  (fun () -> eval e t1)
  | Lam t        -> Value (fun v -> eval (v ::  e) t)

let main (t : term) : value = eval [] t
```

After closure conversion

```
type thunk = Thunk of term * env
and env = thunk list

type value = Closure of term * env

let rec eval (e : env) (t : term) : value =
  match t with
  | Var n        -> let Thunk(t', e') = List.nth e n in
                    eval e' t'
  | App(t0, t1) -> let Closure(t', e') = eval e t0 in
                    eval (Thunk(t1, e)::e') t'
  | Lam t        -> Closure(t, e)

let main (t : term) : value = eval [] t
```

CPS translation

```
type thunk = Thunk of term * env
and env = thunk list

type value = Closure of term * env

let rec eval (e : env) (t : term) (k : value -> 'a) : 'a =
  match t with
  | Var n        -> let Thunk(t', e') = List.nth e n in
                    eval e' t' k
  | App(t0, t1) -> eval e t0 (function Closure(t', e') ->
                    eval (Thunk(t1, e)::e') t' k)
  | Lam t        -> k (Closure(t, e))

let main (t : term) : value = eval [] t (fun x -> x)
```

Defunctionalization of continuations

```ocaml
type thunk = Thunk of term * env
and env = thunk list

type value = Closure of term * env

type stack = thunk list

let rec eval (e : env) (t : term) (s : stack) : value =
  match t, s with
  | Var n,        _   -> let Thunk(t', e') = List.nth e n in
                         eval e' t' s
  | Lam t, v :: s   -> eval (v :: e) t s
  | App(t0, t1), _ -> eval e t0 (Thunk(t1, e) :: s)
  | Lam t, []        -> Closure(t, e)

let main (t : term) : value = eval [] t []
```

Reformatted as abstract machine

```
type thunk = Thunk of term * env
and env = thunk list
type conf = term * env * env

let transition (c:conf) : conf =
  match c with
  | (App(t0, t1), e, s)           -> (t0, e, Thunk(t1, e) :: s)
  | (Lam t, e, v :: s)            -> (t, v :: e, s)
  | (Var 0, Thunk(t, e) :: _, s)  -> (t, e, s)
  | (Var n, _ :: e, s)            -> (Var(n-1), e, s))

let load (t : term) : conf = (t, [], [])
```

Normalization by evaluation

- programming technique to produce full normal forms reduction-free, based on denotational semantics
- interpret terms in a model
- then reify semantic values into syntactic normal forms

Normalization by evaluation for CbV

```
type term  = Var of index | Lam of term | App of term * term

type level = int
type sem = Abs of (sem -> sem) | Neutral of (level -> term)

let to_sem (f : sem -> sem) : sem = Abs f

let from_sem (d : sem) : sem -> sem =
  fun d' ->
    match d with
    | Abs f ->
      f d'
    | Neutral l ->
      Neutral (fun m -> let n = reify d’ m in App (l m, n))
```

Normalization by evaluation for CbV

```
let rec eval (t : term) (e : sem list) : sem =
  match t with
  | Var n -> List.nth e n
  | Lam t' -> to_sem (fun d -> eval t' (d :: e))
  | App (t1, t2) -> let d2 = eval t2 e
                    in from_sem (eval t1 e) d2

let rec reify (d : sem) (m : level) : term =
  match d with
  | Abs f ->
    Lam (reify (f (Neutral (fun m' -> Var (m'-m-1))))(m+1))
  | Neutral l ->
    l m

let nbe (t : term) : term = reify (eval t []) 0
```

Normalization by evaluation for CbV

- functional correspondence applied to CbV NbE produces AM performing full normalization in Strong CbV strategy
- from the machine we can read off the reduction contexts
- obtained AM is inefficient: does not reuse constructed structures and suffers from size explosion

Size explosion problem

$$\omega := \lambda x.\, x\; x$$

$$e_n := \lambda x.\, c_n\; \omega\; x$$

Under Strong CbV

e_n normalizes in linear number of steps

to normal form of exponential size

$$\begin{array}{lll} \omega^1\; x & \to & x\; x \\ \omega^2\; x & \to^* & (x\; x)\; (x\; x) \\ & \vdots & \end{array}$$

NbE for CbV – memoization

```
type 'a cache = 'a option ref

let cached_call (c : 'a cache) (f : unit -> 'a) : 'a =
  match !c with
  | Some y -> y
  | None   -> let y = f () in
              c := Some y;
              y
```

NbE for CbV – memoization

```
type sem = Abs of (sem -> sem)
         | Neutral of (unit -> term)
         | Cache of term cache * sem

let rec from_sem : sem -> (sem -> sem) = function
  | Abs f                   -> f
  |           Neutral l     -> apply_neutral l
  | Cache (c, Neutral l)    -> apply_neutral
                                 (fun () -> cached_call c l)
  | Cache (c,           v)  -> from_sem v
and apply_neutral (l : unit -> term) (v : sem) : sem =
  Neutral (fun () -> let n = reify v in App (l (), n))
```

NbE for CbV – eval and reify

```
let rec eval (t : term) (e : env) : sem =
  match t with
  | Var x          -> env_lookup x e
  | Lam (x, t')    -> to_sem
      (fun v -> eval t' @@ Dict.add x (mount_cache v) e)
  | App (t1, t2) -> let v2 = eval t2 e
                    in from_sem (eval t1 e) v2

let rec reify : sem -> term = function
  | Abs f -> let xm = "x_" ^ string_of_int (gensym ()) in
    Lam (xm, reify (f @@ abstract_variable xm))
  | Neutral l -> l ()
  | Cache (c, v) -> cached_call c (fun () -> reify v)
```

RKNV – abstract machine for SCbV

$$
\begin{array}{rrcl}
\text{Terms} & t & ::= & x \mid t_1\ t_2 \mid \lambda x.\, t \\
\text{Values} & v & ::= & V(x) \mid v_1\ v_2 \mid [x, t, E] \mid v^{\ell} \\
\text{Frames} & F & ::= & [t, E]\,\square \mid \square\ v \mid v\ \square \mid \square\ t \mid \lambda x.\,\square \mid \mathtt{@}[\ell] \\
\text{Heaps} & H & : & \mathtt{location} \to \mathtt{term\ option} \\
\text{Conf.} & K & ::= & \langle t, E, S, m, H\rangle_{\mathcal{E}} \mid \langle S, v, m, H\rangle_{\mathcal{C}} \\
 & & & \mid \langle S, t, m, H\rangle_{\mathcal{S}} \mid \langle t^{?}, \ell, S, v, m, H\rangle_{\mathcal{M}}
\end{array}
$$

RKNV – abstract machine for SCbV

$$\begin{aligned}
\langle [n], \ell, S_2, v, m, H\rangle_{\mathcal{M}} &\rightarrow \langle S_2, n, m, H\rangle_{\mathcal{S}} \\
\langle \bullet, \ell, S_2, v, m, H\rangle_{\mathcal{M}} &\rightarrow \langle @[\ell] :: S_2, v, m, H\rangle_{\mathcal{C}} \\
\langle @[\ell] :: S_2, n, m, H\rangle_{\mathcal{S}} &\rightarrow \langle S_2, n, m, H[\ell := [n]]\rangle_{\mathcal{S}}
\end{aligned}$$

RKNV – sound and complete

- RKNV traces right-to-left strong call by value
- normal forms are equal up to α-equivalence

RKNV – complexity

- amortized cost analysis based on configuration potential
- potential Φ_K of configuration K = how many steps the machine can make till the next β-step
- all but one transitions decrease potential

RKNV – complexity

Configuration potential

$$\Phi_{\mathsf{K}}(K) := \Phi_{\mathsf{t}}(t) + \Phi_{\mathsf{S}}(S) + \Phi_{\mathsf{H}}(K) \qquad \text{if } K = \langle t, E, S, m, H \rangle_{\mathcal{E}}$$

Erasure transition (precedes β-step)

$$\langle \square\ v :: S_1, [x, t, E]^{\ell}, m, H \rangle_{\mathcal{C}} \stackrel{\mathsf{pre}\beta}{\rightarrow} \langle \square\ v :: S_1, [x, t, E], m, H \rangle_{\mathcal{C}}$$

- If $K \stackrel{\neq(\mathsf{pre}\beta)}{\rightarrow} K'$ then $\Phi_{\mathsf{K}}(K) > \Phi_{\mathsf{K}}(K')$
- If $K \stackrel{(\mathsf{pre}\beta)}{\rightarrow} K'$ then $\Phi_{\mathsf{K}}(K) + \Phi_{\mathsf{t}}(\mathsf{input}) > \Phi_{\mathsf{K}}(K')$

RKNV – complexity

reasonability Let: $|\rho|$ – number of transitions starting from term t_0, n – the number of β-reductions in rrSCbV normalization of term t_0.

Then $|\rho| \leq (n+1) \cdot \Phi_t(t_0)$.

overall complexity

$$O((1+n) \cdot |t_0| \cdot E(|t_0|))$$

$E(n)$ – cost of operations on environment of size n

RKNV – complexity

- Strong CbV can be simulated in polynomial time
- Strong CbV calculus is a reasonable time cost model – using approach alternative to [Accattoli et al. '21]

Strong CbNeed – how to approach it

- extend weak CbNd
- two approaches for weak CbNd: storeless or store-based

Strong CbNeed – storeless

- complex declarative definition of reduction semantics [Balabonski et al. '17]
- expressible with generalized reduction semantics with contexts parameterized with sets of variables [Biernacka et al. '19]
- AM for SCbNd derived from generalized reduction semantics by refocusing

Strong Call by Need – example

$\lambda z.\,(\lambda x.\,x\ t)\,(z\ z)$ $\quad$ z is frozen

$\to \lambda z.\,\texttt{let}\ x = z\ z\ \texttt{in}\ x\ t$

Strong Call by Need – example

$\lambda z. (\lambda x. x\ t)\,(z\ z)$ $\quad z$ is frozen

$\to \lambda z. \texttt{let}\ x = z\ z\ \texttt{in}\ x\ t$

$\equiv \lambda z. \texttt{let}\ x = z\ z\ \texttt{in}\ [x]\ t$

Strong Call by Need – example

$\lambda z.\,(\lambda x.\,x\ t)\,(z\ z)$ $\quad z$ is frozen

$\rightarrow \lambda z.\,\texttt{let}\ x = z\ z\ \texttt{in}\ x\ t$

$\equiv \lambda z.\,\texttt{let}\ x = z\ z\ \texttt{in}\ [x]\ t$

$\rightarrow \lambda z.\,\texttt{let}\ x := z\ z\ \texttt{in}\ x\ t$

Strong Call by Need – example

$\lambda z.\,(\lambda x.\, x\ t)\,(z\ z)$ $\quad z$ is frozen

$\rightarrow \lambda z.\, \texttt{let}\ x = z\ z\ \texttt{in}\ x\ t$

$\equiv \lambda z.\, \texttt{let}\ x = z\ z\ \texttt{in}\ [x]\ t$

$\rightarrow \lambda z.\, \texttt{let}\ x := z\ z\ \texttt{in}\ x\ t$

$\equiv \lambda z.\, \texttt{let}\ x := z\ z\ \texttt{in}\ [x]\ t$ $\quad x$ is frozen

Strong Call by Need – example

$\lambda z.\,(\lambda x.\,x\ t)\,(z\ z)$ z is frozen

$\to\ \lambda z.\,\texttt{let}\ x = z\ z\ \texttt{in}\ x\ t$

$\equiv\ \lambda z.\,\texttt{let}\ x = z\ z\ \texttt{in}\ [x]\ t$

$\to\ \lambda z.\,\texttt{let}\ x := z\ z\ \texttt{in}\ x\ t$

$\equiv\ \lambda z.\,\texttt{let}\ x := z\ z\ \texttt{in}\ [x]\ t$ x is frozen

$\equiv\ \lambda z.\,\texttt{let}\ x := z\ z\ \texttt{in}\ x\ [t] \to \ldots$

Strong CbNeed – alternative approach

- starting point: NbE normalizer for normal order
- introduce memoized thunks to avoid recomputation of arguments and of normal forms
- apply functional correspondence to derive AM

Strong CbNeed – RKNL abstract machine

$$\langle [\lambda x.\, t, e], s, \sigma \rangle_{\triangledown} \;\to\; \langle [\lambda x.\, t, e]^{\ell}, s, \sigma * [\ell \mapsto \bot] \rangle_{\vartriangle}$$

$$\langle [\lambda x.\, t, e]^{\ell}, s, \sigma \rangle_{\vartriangle} \;\to\; \langle [t, e[x := \ell_2]], \lambda \check{x}.\, \square :: @[\ell] :: s, \sigma * [\ell_2 \mapsto \check{x}_{\checkmark}] \rangle_{\triangledown}$$

where $\sigma(\ell) = \bot$

$$\langle [\lambda x.\, t, e]^{\ell}, \square\, [t_2, e_2] :: s, \sigma \rangle_{\vartriangle} \to \langle [t, e[x := \ell_2]], s, \sigma * [\ell_2 \mapsto [t_2, e_2]] \rangle_{\triangledown}$$

Strong CbNeed – results

- derived store-based AM
- RKNL simulates the normal-order strategy:
  - each machine configuration accounts for a sequence of NO reduction steps (modulo α-equivalence)
  - each NO reduction step is simulated by a sequence of machine steps
- amenable to complexity analysis using potential function: number of transitions bilinear in the number of β-steps in NO and in size of initial term

A zoo of strategies

- collection of reduction strategies
- characterized by *term decompositions*
- based on ubiquitous *reduction contexts*
- formalized in Coq

Context

```
Inductive frame : Type :=
|  Lam : string → frame
| Rapp : term → frame
| Lapp : term → frame.

Definition context : Type := list frame.
```

Example – CBN contexts

```
Definition CBN_frame (f:frame) : Prop :=
match f with
| Rapp _ => True
| _      => False
end.

Fixpoint Uniform (F:frame → Prop) (C:context) : Prop :=
match C with
| []     => True
| f :: C => F f ∧ Uniform F C
end.

Definition CBN : context → Prop := Uniform CBN_frame.
```

Strategy

```
Definition decomposition : Type := context * term.
Definition strategy      : Type := decomposition → Prop.

Definition recompose : decomposition → term := uncurry plug.

Definition normal_form (s:strategy) (t:term) : Prop :=
  ¬ ∃ d, t = recompose d ∧ d ∈ s.

Definition det_strategy (s:strategy) : Prop :=
  ∀ t, ∃≤1 d, t = recompose d ∧ d ∈ s.
```

Example – CBN strategy

```
Definition β_contrex : term → Prop :=
  app_of abstraction ●.

Definition cbn : strategy :=
  CBN × β_contrex.
```

Normal forms

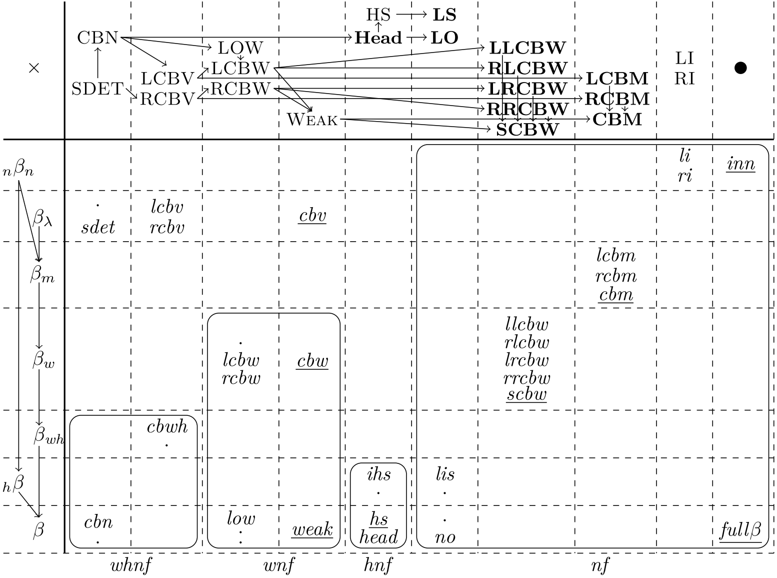
```
Fixpoint rigid (t:term) : Prop :=
match t with
| var _ => True
| app s _ => rigid s
| _ => False
end.

Definition whnf : term → Prop := abstraction ∪ rigid.

Example rigid_is_whnf : rigid ⊆ whnf.

Lemma cbn_nf : normal_form cbn == whnf.
```

Zoo



×
CBN
SDET
LCBV
RCBV
LOW
LCBW
RCBW
WEAK
HS
Head
LS
LO
LLCBW
RLCBW
LRCBW
RRCBW
SCBW
LCBM
RCBM
CBM
LI
RI
$_n\beta_n$
β_λ
β_m
β_w
β_{wh}
$_h\beta$
β
li
ri
inn
sdet
lcbv
rcbv
cbv
lcbm
rcbm
cbm
lcbw
rcbw
cbw
llcbw
rlcbw
lrcbw
rrcbw
scbw
cbwh
ihs
lis
cbn
low
weak
hs
head
no
fullβ
whnf
wnf
hnf
nf

Phased strategies

```
Definition sequence_strategy (r s: strategy) : strategy :=
λ d, r d ∨ (normal_form r (recompose d) ∧ s d).

Notation "↙" := left_strategy.
Notation "'β'" := only_β_contraction.

Definition    cbn_phased := ↙  cbn;; β.

Definition     no_phased := (β;; ↙ no;;  ↘ no) ∪ ↓ no.
```

Phased strategies

```
Definition    cbw_phased := ( ↙ cbw ∪ ↘ cbw);; β.

Definition   scbw_phased := (cbw;; ( ↙ scbw ∪ ↘ scbw)) ∪ ↓scbw.

Lemma scbw_conservative_extension_cbw : scbw == cbw;; scbw.
```

Phased strategies

```
Lemma sequence_strategy_assoc : ∀ q r s,
  q;; (r;; s) == (q;; r);; s.

Lemma left_weak_strategy_β_contraction_commutative : ∀ w,
  w ⊆ weak → ↙ w;; β == β;; ↙ w.

Lemma phased_left_strategy : ∀ s s',
  ↙ s;; ↙ s' == ↙ (s;; s').
```

Benefits

- framework to study, compare and discover new strategies
- more structured and generic proofs of strategy properties, normal forms, etc.
- algebraic reasoning about strategies

Thank you!