Normalization and Partial Evaluation
Lecture 1:
Combinatory Logic and System T

APPSEM 2000
Summerschool
Caminha, Portugal
Summary

• What is traditional normalization?

• What is normalization by evaluation?

• Why “normalization by intuitionistic model construction”?

• A first programming language: a combinatorial version of System T.

• Standard and non-standard model, intuitionistically.

• How to program normalization by evaluation
Reduction

Early proof theory: normalization for logical systems eg natural deduction and sequent calculus. Consistency proofs. (Gentzen, Herbrand?). Lambda calculus.

A notion of “reduction” or simplification of proof or lambda term or combinator. $\text{red}$ is a transitive and reflexive relation.

Reduction rules for combinatory logic:

- $K a b \quad \text{red} \quad a$
- $S a b c \quad \text{red} \quad a c (b c)$
Normalization

$b$ is a normal form iff $b$ is irreducible: $b \text{ red } b'$ implies $b = b'$.

$a$ has normal form $b$ iff $a \text{ red } b$ and $b$ is a normal form.

red is weakly normalizing if all terms have normal form.

red is strongly normalizing if red is a well-founded relation, that is, there is no infinite sequence:

$$a \text{ red } a_1 \text{ red } a_2 \text{ red } \cdots$$

ad infinitum.
Confluence

red is *Church-Rosser* iff a red b and a red b' implies that there is a c such that

\[
\begin{array}{c}
  a \\
  \searrow^{\text{red}} \swarrow^{\text{red}} \\
  b & \text{red} & b' \\
  \searrow^{\text{red}} \swarrow^{\text{red}} \\
  c
\end{array}
\]

Church-Rosser implies uniqueness of normal forms: If a has normal forms b and b', then b = b'.

– Typeset by Foil\TeX –
The decision problem for conversion

Convertibility $\text{conv}$ is the least equivalence relation containing $\text{red}$. Weak normalization plus Church-Rosser of $\text{red}$ yields solution of decision problem for convertibility. (Provided there is an effective strategy which always reaches the normal form.)
A “reduction-free” approach

Start instead with \( \text{conv} \) (no notion of \( \text{red} \)). An abstract normal form function is a function \( \text{norm} \) which picks a canonical representative from each \( \text{conv} \) - equivalence class:

\[
a \text{conv} a' \leftrightarrow \text{norm} a = \text{norm} a'
\]

Decompose it into “existence”

\[
a \text{conv} \text{norm} a
\]

and a “uniqueness”

\[
a \text{conv} a' \rightarrow \text{norm} a = \text{norm} a'
\]

of normal forms. (Nbe is more than normalization; it is normalization + Church-Rosser)
Normalization by evaluation

Normalization by “evaluation” in a model.

\[
\begin{align*}
\text{syntax} & \xrightarrow{\text{model}} [-] \\
\text{reify} & 
\end{align*}
\]

\text{reify} is a left inverse of \([-]\) - the “inverse of the evaluation function” Define

\[
\text{norm } a = \text{reify } [a]
\]

Strictification:

\[
a \text{ conv } a' \rightarrow [a] = [a']
\]

Reification

\[
[A \Rightarrow B] = T(A \Rightarrow B) \times ([A] \rightarrow [B])
\]
\[
[N] = N
\]

\[
\text{reify}_A : [A] \rightarrow T(A)
\]

\[
\text{reify}_{A \Rightarrow B} \langle c, f \rangle = c
\]
\[
\text{reify}_N 0 = \text{ZERO}
\]
\[
\text{reify}_N (s \ p) = \text{APP} (\text{SUCC}, \text{reify}_N p)
\]
The glueing interpretation

\[
\begin{align*}
[A \Rightarrow B] &= T(A \Rightarrow B) \times ([A] \to [B]) \\
[N] &= N
\end{align*}
\]

\[
\llbracket A \rrbracket_A : T(A) \to [A]
\]

\[
\begin{align*}
\llbracket K \rrbracket &= \langle K, \lambda p.\langle \text{APP}(K, \text{reify } p), \lambda q.p \rangle \rangle \\
\llbracket S \rrbracket &= \langle S, \lambda p.\langle \text{APP}(S, \text{reify } p), \langle \ldots, \ldots \rangle \rangle \rangle \\
\llbracket \text{APP}(c, a) \rrbracket &= \text{appsem } \llbracket c \rrbracket \llbracket a \rrbracket \\
\llbracket \text{ZERO} \rrbracket &= 0 \\
\llbracket \text{SUCCE} \rrbracket &= \langle \text{SUCCE}, s \rangle \\
\llbracket \text{REC} \rrbracket &= \langle \text{REC}, \lambda p.\langle \text{APP}(\text{REC}, \text{reify } p), \langle \ldots, \ldots \rangle \rangle \rangle \\
\end{align*}
\]

where

\[
\text{appsem } \langle c, f \rangle q = f q
\]
Correctness proof

\[ a \ \text{conv} \ a' \rightarrow \llbracket a \rrbracket = \llbracket a' \rrbracket \]

is just soundness of interpretation, proved by induction on \( a \ \text{conv} \ a' \).

\[ a \ \text{conv} \ \text{reify} \ \llbracket a \rrbracket \]

is proved by “glueing a la Lafont”. 
In Standard ML

The datatype of syntactic terms

datatype syn = S
  | K
  | APP of syn * syn
  | ZERO
  | SUCC
  | REC

With dependent types we can index the datatype of terms by the object language type.

The reflexive datatype of semantic values:

datatype sem = FUN of syn * (sem -> sem)
  | NAT of int

With dependent types we can use the “universe of metalanguage types” for the interpretation.
Reify

reify : sem -> syn

fun reify (FUN (syn, _))
  = syn
| reify (NAT n)
  = let fun reify_nat 0
      = ZERO
   | reify_nat n
      = APP (SUCC, reify_nat (n-1))
in reify_nat n
end
Evaluation

eval : syn -> sem

fun eval K
    = FUN (K,
        fn x => let val Kx = APP (K, reify
            in FUN (Kx,
                fn _ => x)
            end)
    | eval S
        = FUN (S,
            fn f => let val Sf = APP (S, reify
                in FUN (Sf,
                    ...
                end)
    | eval (APP (e0, e1))
        = appsem (eval e0, eval e1)
    | eval ZERO
        = NAT 0
    | eval SUCC
        = FUN (SUCC,
succsem)
| eval REC
  = FUN (REC,
  fn z
  => let val RECz = APP (REC, reify
  in FUN (RECz,
  ...
  end)
  end)
  end)