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 intutionistic model construction"?
- A first programming language: a combinatory 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. `red` is a transitive and reflexive relation.

Reduction rules for combinatory logic:

$$\begin{array}{rcl} \mathtt{K}\,a\,b & \mathtt{red} & a \\ \mathtt{S}\,a\,b\,c & \mathtt{red} & a\,c\,(b\,c) \end{array}$$

Normalization

b *is a normal form* iff b is irreducible: b `red` b' implies $b = b'$.

a *has normal form* b iff a `red` b and b *is a normal form*.

`red` is *weakly normalizing* if all terms have normal form.

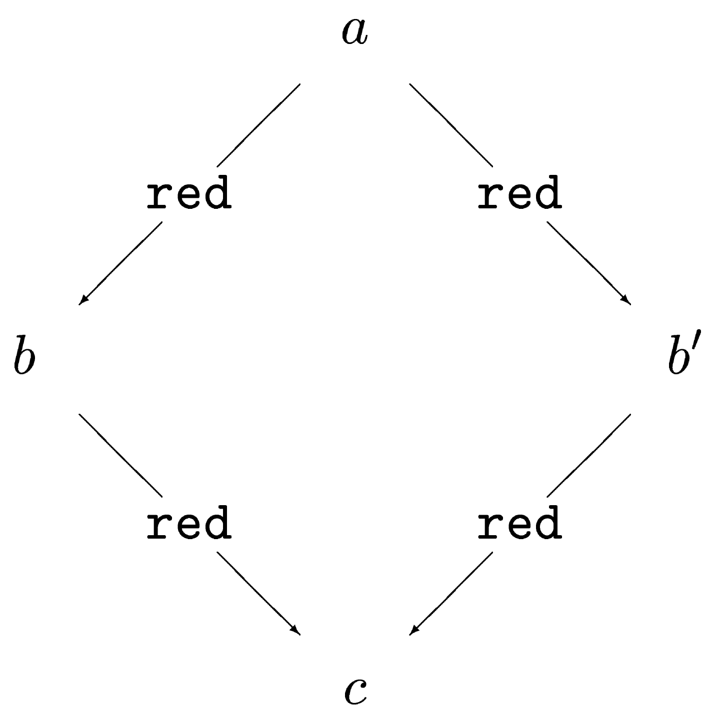
`red` is *strongly normalizing* if `red` is a well-founded relation, that is, there is no infinite sequence:

$$a \;\texttt{red}\; a_1 \;\texttt{red}\; a_2 \;\texttt{red}\; \cdots$$

ad infinitum.

Confluence

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



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

The decision problem for conversion

Convertibility `conv` is the least equivalence relation containing `red` . Weak normalization plus Church-Rosser of `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 `conv` (no notion of `red`). An abstract normal form function is a function **norm** which picks a canonical representative from each `conv` - equivalence class:

$$a \ \texttt{conv} \ a' \leftrightarrow \textbf{norm} \ a = \textbf{norm} \ a'$$

Decompose it into “existence”

$$a \ \texttt{conv} \ \textbf{norm} \ a$$

and a “uniqueness”

$$a \ \texttt{conv} \ a' \rightarrow \textbf{norm} \ a = \textbf{norm} \ a'$$

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

Normalization by evaluation

Normalization by "evaluation" in a model.

$$\text{syntax} \underset{\textbf{reify}}{\overset{[\![-]\!]}{\rightleftarrows}} \text{model}$$

reify is a left inverse of $[\![-]\!]$ - the "inverse of the evaluation function" Define

$$\textbf{norm}\ a = \textbf{reify}\ [\![a]\!]$$

Strictification:

$$a\ \texttt{conv}\ a' \to [\![a]\!] = [\![a']\!]$$

Also, "normalization by intuitionistic model construction". Per Martin-Löf 1975: "About Models for Intuitionistic Type Theories and The Notion of Definitional Equality" - the first paper on normalization by evaluation.

Reification

$$\begin{aligned} [\![A \Rightarrow B]\!] &= \mathtt{T}(A \Rightarrow B) \times ([\![A]\!] \to [\![B]\!]) \\ [\![\mathtt{N}]\!] &= N \end{aligned}$$

$$\textbf{reify}_A \ : [\![A]\!] \to \mathtt{T}(A)$$

$$\begin{aligned} \textbf{reify}_{A \Rightarrow B} \ \langle c, f \rangle &= c \\ \textbf{reify}_{\mathtt{N}} \ 0 &= \mathtt{ZERO} \\ \textbf{reify}_{\mathtt{N}} \ (s \ p) &= \mathtt{APP}(\mathtt{SUCC}, \textbf{reify}_{\mathtt{N}} \ p) \end{aligned}$$

The glueing interpretation

$$\begin{aligned} [\![A \Rightarrow B]\!] &= \mathtt{T}(A \Rightarrow B) \times ([\![A]\!] \to [\![B]\!]) \\ [\![\mathtt{N}]\!] &= N \end{aligned}$$

$$[\![\ ]\!]_A : \mathtt{T}(A) \to [\![A]\!]$$

$$\begin{aligned} [\![\mathtt{K}]\!] &= \langle \mathtt{K}, \lambda p.\langle \mathtt{APP}(\mathtt{K}, \textbf{reify}\ p), \lambda q.p\rangle\rangle \\ [\![\mathtt{S}]\!] &= \langle \mathtt{S}, \lambda p.\langle \mathtt{APP}(\mathtt{S}, \textbf{reify}\ p), \langle \ldots, \ldots\rangle\rangle\rangle \\ [\![\mathtt{APP}(c,a)]\!] &= appsem\ [\![c]\!]\ [\![a]\!] \\ [\![\mathtt{ZERO}]\!] &= 0 \\ [\![\mathtt{SUCC}]\!] &= \langle \mathtt{SUCC}, s\rangle \\ [\![\mathtt{REC}]\!] &= \langle \mathtt{REC}, \lambda p.\langle \mathtt{APP}(\mathtt{REC}, \textbf{reify}\ p), \langle \ldots, \ldots\rangle\rangle\rangle \end{aligned}$$

where

$$appsem\ \langle c, f\rangle\ q = f\ q$$

Correctness proof

$$a \; \texttt{conv} \; a' \to [\![a]\!] = [\![a']\!]$$

is just soundness of interpretation, proved by induction on $a \; \texttt{conv} \; a'$.

$$a \; \texttt{conv} \; \textbf{reify} \; [\![a]\!]$$

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, reif
                   in FUN (Kx,
                           fn _ => x)
                   end)
  | eval S
    = FUN (S,
           fn f => let val Sf = APP (S, reif
                   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)
```