Semantics of Programming Languages (11)

Christian Urban

http://www4.in.tum.de/~urbanc/Teaching/semantics08.html
Beta-reducing lambda-terms can be seen as a form of computation. But this point of view leaves out several important aspects of programming languages.
Beta-reducing lambda-terms can be seen as a form of computation. But this point of view leaves out several important aspects of programming languages.

The idea behind beta-reduction is to pass an argument to a function.
In programming languages, functions themselves are normally not modified, but

\[ \text{Lam } [x].t \xrightarrow{\beta} \text{Lam } [x].t' \]
In programming languages, functions themselves are normally not modified, but

\[ \text{Lam } [x].t \longrightarrow_\beta \text{ Lam } [x].t' \]

Beta-reduction does not specify an interpreter (for example, should the argument be beta-reduced first, or not).
Reduction Reloaded

- In programming languages, functions themselves are normally not modified, but
  \[ \text{Lam}[x].t \rightarrow_\beta \text{Lam}[x].t' \]

- Beta-reduction does not specify an interpreter (for example, should the argument be beta-reduced first, or not).

- We want to improve programming (that is help the programmer by providing better programming languages).
Functions as Values

inductive

beta_red :: "lam⇒lam⇒bool" ("_ →_β_"")

where

beta: "App (Lam [x].t) t' →_β t[x::=t']"

app1: "t →_β t' ⇔ App t t2 →_β App t' t2"

app2: "t →_β t' ⇔ App t2 t →_β App t2 t'"

lam: "t →_β t' ⇔ Lam [x].t →_β Lam [x].t"

We like to disallow reduction under lambdas.
We like to disallow reduction under lambdas.
inductive

\texttt{beta\_red :: "lam\Rightarrow lam\Rightarrow bool" ("_ \longrightarrow_\beta _")}

\texttt{where}

\texttt{beta: "App (Lam \[x\].t) t' \longrightarrow_\beta t[x::=t']"}

\texttt{app1: "t \longrightarrow_\beta t' \quad \Longrightarrow \quad App t t2 \longrightarrow_\beta App t' t2"}

\texttt{app2: "t \longrightarrow_\beta t' \quad \Longrightarrow \quad App t2 t \longrightarrow_\beta App t2 t"}

\textbf{All three rules overlap.}
Eliminating Non-Determinism

\[ \text{inductive} \]
\[ \text{beta_red :: } "\text{lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" \ ("_ \rightarrow_\beta _") \]

\[ \text{where} \]
\[ \text{beta: } "\text{App (Lam [x].t) t'} \rightarrow_\beta t[x::=t']" \]
\[ \text{app1: } "t \rightarrow_\beta t' \Rightarrow \text{App t t2} \rightarrow_\beta \text{App t'} t2" \]
\[ \text{app2: } "t \rightarrow_\beta t' \Rightarrow \text{App t2 t} \rightarrow_\beta \text{App t2 t'}" \]

- All three rules overlap.

- \text{call-by-value}: before an argument is passed to a function, it is reduced to normal form

- \text{call-by-name}: arguments are directly substituted into a function

- \text{call-by-need}: if a function needs an argument, it will be reduced and the result is stored for future uses
Reduction Strategy

- **call-by-value**: reduces arguments once
- **call-by-name**: reduces arguments as often as they are used
- **call-by-need**: reduces arguments at most once
Reduction Strategy

- **call-by-value**: reduces arguments once
- **call-by-name**: reduces arguments as often as they are used
- **call-by-need**: reduces arguments at most once

- call-by-value is most commonly used strategy (e.g. SML, Scheme)
- call-by-need is used in Haskell
- call-by-name and call-by-need can cause problems with non-termination... \(((\lambda x. x \ x)(\lambda x. x \ x))\)
inductive

\(vbeta\_red:: "\text{lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" ("_ \rightarrow_\beta v _")\)

where

\(vbeta:: "\text{App (Lam \([x].t) t' } \rightarrow_\beta v \ t[x::=t']"\)

\(vapp1:: "\text{[val } t_2; t \rightarrow_\beta v t'] \Rightarrow \text{App } t \ t_2 \rightarrow_\beta v \text{ App } t' \ t_2"\)

\(vapp2:: "t \rightarrow_\beta v t' \Rightarrow \text{App } t_2 \ t \rightarrow_\beta v \text{ App } t_2 \ t'"\)
Call-by-value

inductive

\[ \text{vbeta_red} :: "\text{lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" \ ("_ \rightarrow^\beta v _") \]

where

\[ \text{vbeta} : "\text{App (Lam } [x].t) t' \rightarrow^\beta v t[x::=t']" \]
\[ \text{vapp1} : "[\text{val } t_2; t \rightarrow^\beta v t'] \rightarrow \text{App } t \ t_2 \rightarrow^\beta v \text{ App } t' \ t_2" \]
\[ \text{vapp2} : "t \rightarrow^\beta v t' \rightarrow \text{App } t_2 \ t \rightarrow^\beta v \text{ App } t_2 \ t'" \]

definition

\[ \text{val} :: "\text{lam } \Rightarrow \text{bool}" \]

where

"\text{val } t \equiv \neg (\exists \ t'. \ t \rightarrow^\beta v \ t')"
**Call-by-value**

**inductive**

\[ \text{vbeta\_red :: "lam} \Rightarrow \text{lam} \Rightarrow \text{bool" ("}_ \beta \text{v } ") \]

**where**

\[ \text{vbeta: "App (Lam [x].t) t'} \beta v t[x::=t']" } \]
\[ \text{vapp1: "[val t_2; t} \beta v t'} \beta v \text{ App t'} t_2" } \]
\[ \text{vapp2: "t} \beta v t'} \beta v \text{ App t_2 t'}" } \]

**inductive**

\[ \text{val :: "lam} \Rightarrow \text{bool" } \]

**where**

\[ "\text{val (Var x)}" \]
\[ "\text{val (Lam [x].t)}" \]
So far we considered small-step reductions. Here is a definition of a big-step reduction, or big-step evaluation.

**inductive**

\[
\text{eval} :: "\text{lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" ("\_ \Downarrow \_")
\]

**where**

\[
\begin{align*}
\text{eval}_{\text{var}} & : "\text{Var } x \Downarrow \text{Var } x" \\
\text{eval}_{\text{lam}} & : "\text{Lam } [x].t \Downarrow \text{Lam } [x].t" \\
\text{eval}_{\text{app}} & :
  "\left[ t_1 \Downarrow \text{Lam } [x].t; t_2 \Downarrow t_2'; t[x::=t_2'] \Downarrow t' \right] \Rightarrow \text{App } t_1 t_2 \Downarrow t''"
\end{align*}
\]
Equivalence of $\rightarrow^\beta v^*$ and $\downarrow$

**Inductive**

\[
vbeta\_red :: \text{"lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" \ ("_ \ \rightarrow^\beta v \ _")
\]

**Where**

\[
vbeta : "\text{App (Lam } [x].t) \ t' \ \rightarrow^\beta v \ t[x::=t']"
\]
\[
vapp1 : "[\text{val } t_2; \ t \ \rightarrow^\beta v \ t'] \ \Rightarrow \ \text{App } t \ t_2 \ \rightarrow^\beta v \ \text{App } t' \ t_2"
\]
\[
vapp2 : "t \ \rightarrow^\beta v \ t' \ \Rightarrow \ \text{App } t_2 \ t \ \rightarrow^\beta v \ \text{App } t_2 \ t"
\]

**Inductive**

\[
eval :: \text{"lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" \ ("_ \ \downarrow \ _")
\]

**Where**

\[
eval\_var : "\text{Var } x \ \downarrow \ \text{Var } x"
\]
\[
eval\_lam : "\text{Lam } [x].t \ \downarrow \ \text{Lam } [x].t"
\]
\[
eval\_app:
  "[t_1 \ \downarrow \ \text{Lam } [x].t; \ t_2 \ \downarrow \ t_2'; \ t[x::=t_2'] \ \downarrow \ t'] \ \Rightarrow \ \text{App } t_1 \ t_2 \ \downarrow \ t"
\]
Two Little Lemmas

lemma eval_val:
  assumes a: "val t"
  shows "t \downarrow t"
using a by (induct) (auto)

lemma eval_to:
  assumes a: "t \downarrow t'"
  shows "val t'"
using a by (induct) (auto)
lemma
assumes a: "t ⊢ t'"
shows "t →_β v* t'"
using a
proof (induct)
  case (eval_app t₁ ⨉ t t₂ t₂' t')
  from 't₂ ⊢ t₂' have v: "val t₂''" by (simp add: eval_to)
  have ih₁: "t₁ →_β v* Lam [x].t" by fact
  have ih₂: "t₂ →_β v* t₂''" by fact
  have ih₃: "t[x::=t₂'] →_β v* t''" by fact
  have "App t₁ t₂ →_β v* App t₁ t₂''" using ih₂ by auto
  also have "... →_β v* App (Lam [x].t) t₂''" using ih₁ v by auto
  also have "... →_β v* t[x::=t₂']" using v by auto
  also have "... →_β v* t''" using ih₃ by simp
  finally show "App t₁ t₂ →_β v* t''" by simp
qed (auto)
The definition of big-step and small-step reduction relations determine how lambda-terms are reduced.

It can be shown that they are equivalent (when a normalform is reached).

Small-step reduction can talk about infinite reduction sequences.