CS 520

Theory of Programming Language

05/26 – 06/02, 2021

$\boxed{\text{An Eager Functional Language (Chap 11)}}$

1. Motivation.

1. Lambda Calculus with eager evaluation $\Rightarrow_E$ ..... Ocaml, Scala, Clojure, Scheme.

(Java, Python, C#).

But not convenient for programming.

② (1) Support for primitive operations and basic data types.
(2) support for recursion.

needed in a real-world functional PL.

③ tricky in eager functional PLs.

(1) $f : D \to_c D$ ..... $\exists$ least fixed point of f.

(2) EFL ... eager functional lang.

ⓐ $f : D \to_c D$ ... expressible in EFL.
$D = (V)_\perp$ strict. $f(\perp) = \perp$.

not necessarily a domain ..... $V_0 \subseteq V$.

ⓑ $g : V \to V$ ... what we want.

$V \to V_\perp$ .. general fun. in EFL.
$\hat{D}$

→ ⓐ κ e sumcase e of
τ. $(e_0, \cdots, e_{n-1})$

2. Support for primitive operations and data types.

① $\langle exp\rangle ::= \langle var\rangle \mid \lambda\langle var\rangle.\langle exp\rangle \mid \langle exp\rangle\ \langle exp\rangle. \mid \langle\langle exp\rangle,\dots,\langle exp\rangle\rangle \mid \langle exp\rangle.\langle tag\rangle.$

$\mid @\ \langle tag\rangle\ \langle exp\rangle \mid \text{sumcase}\ \langle exp\rangle\ \text{of}\ (\langle exp\rangle,\dots,\langle exp\rangle)$

$\mid \text{true} \mid \text{false} \mid -2 \mid -1 \mid \dots \mid \text{if}\ \langle exp\rangle\ \text{then}\ \langle exp\rangle\ \text{else}\ \langle exp\rangle.$

$\mid \langle exp\rangle + \langle exp\rangle \mid \langle exp\rangle \wedge \langle exp\rangle \mid$

$\langle cfm\rangle ::= \langle funcfm\rangle. \mid \langle tuplecfm\rangle. \mid \langle altcfm\rangle. \mid \langle boolcfm\rangle \mid \langle intcfm\rangle.$

$\langle tag\rangle ::= 0 \mid 1 \mid 2 \mid \dots$ $\quad k<n.$

$\langle funcfm\rangle ::= \lambda\langle var\rangle.\langle exp\rangle$

$\langle tuplecfm\rangle ::= \langle\langle cfm\rangle,\dots,\langle cfm\rangle\rangle$

$\langle altcfm\rangle ::= @\ \langle tag\rangle\ \langle cfm\rangle$

$\langle boolcfm\rangle ::= \text{true} \mid \text{false}$

$\langle intcfm\rangle ::= \dots -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$

(construct, then destruct = identity).

cfm:

$$\frac{}{z \Rightarrow_E z}$$

β_E-evaluation:

$$\frac{e_1 \Rightarrow_E \lambda x.e_1' \quad e_2 \Rightarrow_E z_2 \quad e_1'/x \to z_2 \Rightarrow_E z.}{e_1\, e_2 \Rightarrow_E z}$$

β-rule tuple:

$$\frac{e \Rightarrow \langle z_0,\dots,z_{n-1}\rangle}{e.k \Rightarrow z_k} \quad (k<n)$$

$$\frac{e_0 \Rightarrow z_0 \quad e_1 \Rightarrow z_1 \quad \cdots \quad e_{n-1} \Rightarrow z_{n-1}}{\langle e_0,\dots,e_{n-1}\rangle \Rightarrow \langle z_0,\dots,z_{n-1}\rangle}$$

β_{alt}-rule.

$$\frac{e \Rightarrow @\, k\, z \qquad e_k(z) \Rightarrow z'}{\text{sumcase}\ e\ \text{of}\ (e_0,\dots,e_{n-1}) \Rightarrow z'.}$$

$$\frac{e \Rightarrow z}{@\, k\, e \Rightarrow @\, k\, z.}$$

$\mathbb{Z}. \quad \mathbb{Z}.$

$$\frac{e_0 \Rightarrow i_0 \quad e_1 \Rightarrow i_1}{e_0 + e_1 \Rightarrow i_0 \hat{+} i_1}$$

$$\frac{e \Rightarrow \text{true}. \quad e_0 \Rightarrow z_0}{\text{if}\ e\ \text{then}\ e_0\ \text{else}\ e_1 \Rightarrow z_0.}$$

$$\frac{e \Rightarrow \text{false} \quad e_1 \Rightarrow z_1}{\text{if}\ e\ \text{then}\ e_0\ \text{else}\ e_1 \Rightarrow z_1}$$

② Principles overall plan:

(1) Extend each of the three components from above.

(2) Think about run-time type and corresponding constructors and destructors.

(3) Add a new case to <cfun> to account for the new run-time type.
Use constructors to define the case.

(4) Extend <exp> with both const. and dest.

(5) Add 2 rules (or more) to ⇒, one for constructor and the other for destructor.

③ Add a tuple data type.

$\langle 3, 4, 5 \rangle$

$\langle 3, 4, e \rangle . 0$

projection

e.1

projection of the 2nd component.

④ Add a data type for alternatives.

@ 0 true , @ 1 ⟨3,4⟩ , @ 2 $(\lambda x.x)$. ... constructor

sumcase e of $(e_0, \dots, e_{n-1})$.

↑

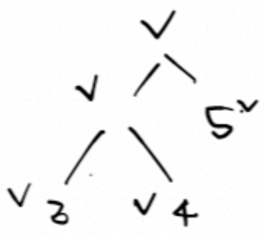
@ 0 true. ⇝ e_0(true) .....

@ 1 ⟨3,4⟩ ---→ e_1 ⟨3,4⟩ ...

true

[@ 0 3 ..... leaf. @ 1 ⟨l, r⟩ ... node.

integer. tree tree.



..... @ 1 ⟨@ 1 ⟨@ 0 3, @ 0 4⟩, @ 0 5⟩.

$\lambda x.$ sumcase x of $(\lambda i.\ \text{true},\ \lambda t.\ \text{false})$

ex. Extend the lang. to support this alternative type.

sum.

5 Support for primitive values and operations about them:

booleans and integers. $+, \wedge, \ldots$

3. Recursion

$$\langle exp \rangle ::= \ \dots \ \mid \ \mathrm{letrec}\ \langle var \rangle = \overbrace{\underbrace{\lambda \langle var \rangle . \langle exp \rangle}_{\text{fn.}}}^{\text{canonical form}}\ \mathrm{in}\ \langle exp \rangle$$

$$FV(\mathrm{letrec}\ x = \lambda y.\, e\ \mathrm{in}\ e') = \big((FV(e) \setminus \{y\}) \cup FV(e') \big) \setminus \{x\}.$$

$$\begin{aligned} &\mathrm{letrec}\ \ add = \lambda t.\ \mathrm{sumcase}\ t\ \mathrm{of} \\ &\qquad\qquad (\lambda i.\, i\ ,\ \lambda n.\ (add\ (n.0)) + (add\ (n.1)))\quad \mathrm{in} \\ &add. \end{aligned}$$