Australian National University

COMP3610/6361
Principles of Programming Languages

Peter Höfner

Aug 16, 2023

Section 7

Recursion

Scoping

Name Definitions
restrict the scope of variables

$$E ::= \ldots \mid \textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end}$$

- x is a binder for E_2
- can be seen as syntactic sugar:

$$\textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end} \quad \equiv \quad (\textbf{fn } x : T \Rightarrow E_2)\ E_1$$

Derived sos-rules and typing

$$\textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end} \equiv (\textbf{fn } x : T \Rightarrow E_2)\ E_1$$

(let)
$$\frac{\Gamma \vdash E_1 : T \qquad \Gamma, x : T \vdash E_2 : T'}{\Gamma \vdash \textbf{let val } x : T = E_1 \textbf{ in } E_2 \textbf{ end} : T'}$$

(let1)
$$\frac{\langle E_1\ ,\ s \rangle \longrightarrow \langle E_1'\ ,\ s' \rangle}{\langle \textbf{let val } x{:}T = E_1 \textbf{ in } E_2 \textbf{ end}\ ,\ s \rangle \longrightarrow \langle \textbf{let val } x{:}T = E_1' \textbf{ in } E_2 \textbf{ end}\ ,\ s' \rangle}$$

(let2)
$$\langle \textbf{let val } x : T = v \textbf{ in } E_2 \textbf{ end}\ ,\ s \rangle \longrightarrow \langle \{v/x\}\, E_2\ ,\ s \rangle$$

Recursion – An Attempt

Consider

$$r = (\textbf{fn}\ y : \text{int} \Rightarrow \textbf{if}\ y \geq 1\ \textbf{then}\ y + (r(y + -1))\ \textbf{else}\ 0)$$

where r is the recursive call (variable occurring in itself).
What is the evaluation of $r\ 3$?

We could try

$$E ::= \ldots \mid \textbf{let val rec}\ x : T = E\ \textbf{in}\ E'\ \textbf{end}$$

where x is a binder for both E and E'.

$$\begin{array}{l} \textbf{let val rec}\ r : \text{int} \to \text{int}\ = \\ \qquad (\textbf{fn}\ y : \text{int} \Rightarrow \textbf{if}\ y \geq 1\ \textbf{then}\ y + (r(y + -1))\ \textbf{else}\ 0) \\ \textbf{in}\ r\ 3\ \textbf{end} \end{array}$$

However . . .

- What about **let val rec** $x : T = (x, x)$ **in** x **end**?
- What about **let val rec** x : int list $= 3 :: x$ **in** x **end**?
  Does this terminate? and if it does is it equal to
  – **let val rec** x : int list $= 3 :: 3 :: x$ **in** x **end**
- Does **let val rec** x : int list $= 3 :: (x + 1)$ **in** x **end** terminate?
- In Call-by-Name (Call-by-Need) these are reasonable
- In Call-by-Value these would usually be disallowed

Recursive Functions

Idea specialise the previous **let val rec**

- $T = T_1 \to T_2$ (recursion only at function types)
- $E = (\textbf{fn}\ y : T_1 \Rightarrow E_1)$ (and only for function values)

Recursive Functions – Syntax and Typing

$E ::= \ldots \mid$ **let val rec** $x : T_1 \to T_2 = ($**fn** $y : T_1 \Rightarrow E_1)$ **in** E_2 **end**

Here, y binds in E_1 and x bind in $($**fn** $y : T_1 \Rightarrow E_1)$ and E_2

$$\text{(recT)} \quad \frac{\Gamma,\ x{:}T_1 \to T_2,\ y{:}T_1 \vdash E_1 : T_2 \qquad \Gamma, x{:}T_1 \to T_2 \vdash E_2 : T}{\Gamma \vdash \textbf{let val rec}\ x : T_1 \to T_2 = (\textbf{fn}\ y : T_1 \Rightarrow E_1)\ \textbf{in}\ E_2\ \textbf{end} : T}$$

Recursive Functions – Semantics

(rec) $\langle \textbf{let val rec } x{:}T_1 \to T_2 = (\textbf{fn } y{:}T_1 \Rightarrow E_1) \textbf{ in } E_2 \textbf{ end}, s\rangle$

$$\longrightarrow$$

$\langle \{(\textbf{fn } y{:}T_1 \Rightarrow \textbf{ let val rec } x{:}T_1 \to T_2 = (\textbf{fn } y{:}T_1 \Rightarrow E_1) \textbf{ in } E_1 \textbf{ end})/x\}\, E_2, s\rangle$

Redundancies?

- Do we need $E_1 \; ; E_2$?
  No: $E_1 \; ; E_2 \equiv (\textbf{fn } y : \text{unit} \Rightarrow E_2) \; E_1$

- Do we need **while** E_1 **do** E_2?
  No:

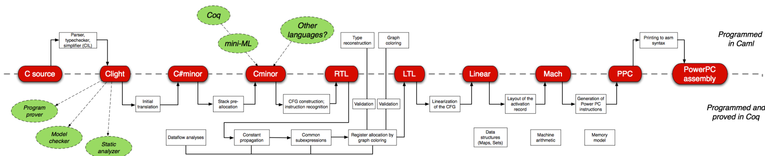
$$
\begin{aligned}
\textbf{while } E_1 \textbf{ do } E_2 \equiv \; & \textbf{let val rec } w : \text{unit} \to \text{unit} \; = \\
& \qquad (\textbf{fn } y : \text{unit} \Rightarrow \textbf{if } E_1 \textbf{ then } (E_2; (w \textbf{ skip})) \textbf{ else skip}) \\
& \textbf{in} \\
& \qquad w \textbf{ skip} \\
& \textbf{end}
\end{aligned}
$$

Redundancies?

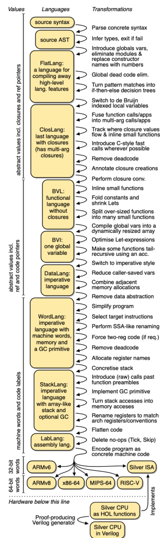
- Do we need recursion?
  Yes! Previously, normalisation theorem effectively showed that **while** adds expressive power; now, recursion is even more powerful.

Side remarks I

- naive implementations (in particular substitutions) are inefficient (more efficient implementations are shown in courses on compiler construction)
- more concrete – closer to implementation or machine code – are possible
- usually refinement to prove compiler to be correct (e.g. CompCert or CakeML)



Side remarks I – CakeML



Side remarks II: Big-step Semantics

- we have seen a **small-step semantics**

$$\langle E\,,\,s\rangle \longrightarrow \langle E'\,,\,s'\rangle$$

- alternatively, we could have looked at a **big-step semantics**

$$\langle E\,,\,s\rangle \Downarrow \langle E'\,,\,s'\rangle$$

For example

$$\frac{}{\langle n\,,\,s\rangle \Downarrow \langle n\,,\,s\rangle} \qquad \frac{\langle E_1\,,\,s\rangle \Downarrow \langle n_1\,,\,s'\rangle \quad \langle E_2\,,\,s'\rangle \Downarrow \langle n_2\,,\,s''\rangle}{\langle E_1 + E_2\,,\,s\rangle \Downarrow \langle n\,,\,s''\rangle}(n = n_1 + n_2)$$

- no major difference for sequential programs
- small-step much better for modelling concurrency and proving type safety