Extending While

Marco Patrignani

1 Letin

$$\textit{statements } s ::= \cdots \mid \textit{let } x = a \textit{ in } s$$

The "letin" construct defines a new fresh variable x with scope s. There is no need to track x in the environment because the substitution does this automatically.

$$\frac{\overset{\text{(While-letin)}}{\Sigma \vdash a \Downarrow n}}{\Sigma \triangleright \textit{let } x = a \textit{ in } s \hookrightarrow \Sigma \triangleright s[n \,/\, x]}$$

Substition:

- capture-avoiding: assume all names are distinct
- defined inductively on statements and expressions. We report only the interesting cases.

$$\begin{aligned} x[n \,/\, y] &= n && \text{if } y = x \\ x[n \,/\, y] &= x && \text{if } y \neq x \\ (x := a)[n \,/\, y] &= (x := a[n \,/\, y]) \end{aligned}$$

  No overriding of left hand sides.

Letin lets us model local scope unlike $x := a$ where all xs are globally available.

2 Named Functions

$$\begin{aligned} \textit{statements} &\quad s ::= \cdots \mid \text{call } f\ a \\ \textit{programs} &\quad P ::= \varnothing \mid P; f(x) \mapsto s \\ \textit{prog. states} &\quad \Omega ::= \Sigma; P \triangleright s \\ \textit{op. sem. judgements} &\quad \Omega \hookrightarrow \Omega \end{aligned}$$

We need to change the notion of program state to account for a lookup table where to find function implementations.

$$\frac{\Sigma \vdash a \Downarrow n \quad f(x) \mapsto s \in P}{\Sigma; P \triangleright \text{call } f\ a \hookrightarrow \Sigma; P \triangleright s[n\ /\ x]}\ \text{(While-call)}$$

The program state collects all that is necessary for the program to run.

- What does it mean to run a program now? Before we could just execute the program, it was just s. Now?
- We need a notion of main.
- Also, we need to ensure that only defined functions are called. You cannot enforce this at grammar level purely: you need a (very simple) form of program analysis to do this. For now we assume this though and leave this analysis for later.

Example 2.1 (Call reduction).

$$\begin{aligned} P = \ & \text{succ}(x) \mapsto y := x + 1 \\ & \text{main}(z) \mapsto y := 0; \text{call } \mathit{succ}\ 3; \text{call } \mathit{succ}\ y; \end{aligned}$$

$$\begin{aligned} & \Sigma; P \triangleright \text{call } \mathit{main}0 \\ \hookrightarrow & \Sigma; P \triangleright (y := 0; \text{call } \mathit{succ}\ 3; \text{call } \mathit{succ}\ y;)[0\ /\ z] \\ \equiv & \Sigma; P \triangleright y := 0; \text{call } \mathit{succ}\ 3; \text{call } \mathit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 0]; P \triangleright \text{skip}; \text{call } \mathit{succ}\ 3; \text{call } \mathit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 0]; P \triangleright \text{call } \mathit{succ}\ 3; \text{call } \mathit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 0]; P \triangleright (y := x + 1)[3\ /\ x]; \text{call } \mathit{succ}\ y; \\ \equiv & \Sigma[y \mapsto 0]; P \triangleright y := 3 + 1; \text{call } \mathit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 4]; P \triangleright \text{skip}; \text{call } \mathit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 4]; P \triangleright \text{call } \mathit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 4]; P \triangleright y := x + 1[4\ /\ x]; \\ \equiv & \Sigma[y \mapsto 4]; P \triangleright y := 4 + 1; \\ \hookrightarrow & \Sigma[y \mapsto 5]; P \triangleright \text{skip}; \end{aligned}$$

⊡

How to define equivalences then? Force the main to be defined in the observer, outside the programs that are to be equated.

3 Breaking Returns

$$\textit{statements} \qquad s ::= \cdots \mid \text{ret}$$

$$\textit{runtime statements} \qquad s ::= \cdots \mid \text{end}$$

Runtime statements are statements that are not part of the grammar, the programmer cannot write them. However, the semantics use them in order to keep track of particular things, in this case of the scope of function bodies.

$$\frac{\overset{\text{(While-call-2)}}{\Sigma \vdash a \Downarrow n} \quad f(x) \mapsto s \in P}{\Sigma; P \triangleright \text{call } f\ a \hookrightarrow \Sigma; P \triangleright s[n\ /\ x]; \text{end}} \qquad \frac{\overset{\text{(While-end)}}{}}{\Sigma; P \triangleright \text{end} \hookrightarrow \Sigma; P \triangleright \text{skip}}$$

$$\frac{\overset{\text{(While-ret)}}{\text{end} \notin s}}{\Sigma; P \triangleright \text{ret}; s; \text{end} \hookrightarrow \Sigma; P \triangleright \text{skip}}$$

Example 3.1 (Returns).

$$\begin{aligned} P = \ & \text{loop}(x) \mapsto \text{while}(x > 0)\{y := y + 1; \text{if}(y > 10) \text{ then ret else } x := x + 1\} \\ & \text{main}(z) \mapsto y := 11; \text{call } \textit{loop}\ 2; y := 8; \end{aligned}$$

We omit skipping reductions and perform substitions right away.

$$\begin{aligned} & \Sigma; P \triangleright \text{main}(0) \\ \hookrightarrow & \Sigma; P \triangleright y := 11; \text{call } \textit{loop}\ 2; y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 11]; P \triangleright \text{call } \textit{loop}\ 2; y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 11]; P \triangleright \text{while}(2 > 0)\{y := y + 1; \text{if}(y > 10) \text{ then ret else } x := x + 1\}\text{end}; y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 11]; P \triangleright\ y := y + 1; \text{if}(y > 10) \text{ then ret else } x := x + 1; \\ & \qquad \text{while}(2 > 0)\{y := y + 1; \text{if}(y > 10) \text{ then ret else } x := x + 1\} \\ & \qquad \text{end}; y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 12]; P \triangleright\ \text{if}(y > 10) \text{ then ret else } x := x + 1; \\ & \qquad \text{while}(2 > 0)\{y := y + 1; \text{if}(y > 10) \text{ then ret else } x := x + 1\} \\ & \qquad \text{end}; y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 11]; P \triangleright\ \text{ret} \\ & \qquad \text{while}(2 > 0)\{y := y + 1; \text{if}(y > 10) \text{ then ret else } x := x + 1\} \\ & \qquad \text{end}; y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 11]; P \triangleright\ \text{skip}; y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 11]; P \triangleright\ y := 8; \\ \hookrightarrow & \Sigma[y \mapsto 8]; P \triangleright\ \text{skip} \end{aligned}$$

⊡

4 Returning Values

$$\textit{statements} \qquad s ::= \cdots \mid \text{ret } a \mid \text{let } x = \text{call } f\ a$$

$$\textit{prog. states} \qquad \Omega ::= \Sigma; P; \overline{x} \triangleright s$$

We need a special construct that tells us how to call a function and store its returned value. Then we need to keep track of the names of the variables for returned values in the program state

- Note: we need a list of such variables because of nested calls, a single variable name would not work!

$$\frac{\Sigma \vdash a \Downarrow n \qquad f(x) \mapsto s \in P}{\Sigma; P; \overline{x} \triangleright \text{let } y = \text{call } f\ a \hookrightarrow \Sigma; P; \overline{x}, y \triangleright s[n\ /\ x]; \text{end}}\ \text{(While-call-3)}$$

$$\frac{\text{end} \notin s \qquad \Sigma \vdash a \Downarrow n}{\Sigma; P; \overline{x}, x \triangleright \text{ret } a; s; \text{end} \hookrightarrow \Sigma; P \triangleright x := n}\ \text{(While-ret-2)}$$

We cannot evaluate call $f\ a$ with the big step semantics of expressions because function bodies are impure statements. So we need a construction to remember where to store the result of the call, and this is the $\overline{x}$ addition to the program state.

Example 4.1 (Returning values).

$$\begin{aligned} P = \ & \text{succ}(x) \mapsto \text{ret } x + 1 \\ & \text{main}(z) \mapsto \text{let } y = \text{call } \textit{succ}\ 3; \text{let } y = \text{call } \textit{succ}\ y; \end{aligned}$$

$$\begin{aligned} & \Sigma; P; \varnothing \triangleright \text{call } \textit{main}0 \\ \hookrightarrow & \Sigma; P; \varnothing \triangleright \text{let } y = \text{call } \textit{succ}\ 3; \text{let } y = \text{call } \textit{succ}\ y; \\ \hookrightarrow & \Sigma; P; y \triangleright \text{ret } 3 + 1; \text{end}; \text{let } y = \text{call } \textit{succ}\ y; \\ \hookrightarrow & \Sigma; P; \varnothing \triangleright y := 4; \text{let } y = \text{call } \textit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 4]; P; \varnothing \triangleright \text{let } y = \text{call } \textit{succ}\ y; \\ \hookrightarrow & \Sigma[y \mapsto 4]; P; y \triangleright \text{ret } 4 + 1 \\ \hookrightarrow & \Sigma[y \mapsto 4]; P; \varnothing \triangleright y := 5 \\ \hookrightarrow & \Sigma[y \mapsto 5]; P; \varnothing \triangleright \text{skip} \end{aligned}$$

⊡