Lecture 5: Variables and Binding

Programming Languages (H)

Simon Fowler & Michele Sevegnani

Semester 2, 2024/2025

sli.do 1275413

University of Glasgow

sli.do 1275413

Overview

- **Last time**
  - Operational semantics
  - Booleans & conditionals
  - Introduction to ANTLR & SVM
- **This lecture:** Variables and binding
  - Let-expressions
  - Binding and scope
  - Abstract binding diagrams
- **Please come to the labs this afternoon!** (15:00 – 17:00 in BO722)
  - We're happy to help, even with last week's lab if you didn't get chance to do it
  - Even if you can't make it, please do have a go; we will answer questions on Teams

sli.do 1275413

Where we're up to

L_{If} Abstract Syntax

```
Integers n
Booleans b     ::= true | false
Operators ⊙   ::= + | - | *  | /
                 | < | > | && | ||
                 | ==
Values V, W    ::= n | b
Constants c    ::= n | b
Terms L, M, N  ::= c
                | L ⊙ M
                | if L then M else N
```

- Last time, we looked at L_{If}, an extension of L_{Arith} with support for Booleans and conditionals
- We also saw **operational semantics**, which allowed us to specify the meaning of evaluating an expression down to a value, and saw how there was a close correspondence between the theory and how an interpreter can be implemented

sli.do 1275413

Where next?

- We are getting *closer* to the core of a useful programming language, but are really limited by the fact we don't have **variables** yet: we can only talk about expressions where all operands are known, e.g.

```
if 1 + 2 < 3 then true && false else true || false
```

- Variable binding has a lot of subtlety: it is easy to get wrong unless you understand the specifics
  - For example, JavaScript's var scoping rules are notoriously complex
- In this and the next lecture we'll take a principled look at binding

sli.do 1275413

You will likely have already seen variable binding in AF2 / Practical Algorithms

Quantifiers – binding and scope

Variables can be bound through quantifiers

- as we have seen unbound variables are also called free variables

A variable x is bound to quantifier $\forall x$ or $\exists x$ if

- it appears free within the scope of the quantifier

Examples: $\forall x.(P(y) \wedge Q(x))$ $\exists x.\forall y.(R(y,x) \wedge Q(x))$

If a quantifier does not bind any variables it can be removed

Example: $\forall y.\exists x.P(x)$

- since y is not a free variable in $\exists x.P(x)$, the "$\forall y$" quantifier is redundant

sli.do 1275413

Recap: Predicate Logic Quantifiers

- In predicate logic we have two **quantifiers**: these allow us to allow a variable to 'stand for' an element of a set
  - Universal quantifiers require a property to hold for all elements of a set
  - Existential quantifiers require a property to hold for at least one element of a set

$$\underbrace{\exists \boxed{c} \in Cities.}_{\text{Quantifier}}\ \underbrace{isFriday \rightarrow Rainy(\boxed{c})}_{\text{Formula body}}$$

"There is some city where it rains every Friday"

Binding occurrence of c

Bound occurrence of c

- A variable is **free** if it is not in the scope of a quantifier. A quantifier $\forall x\,.\,P$ binds all free occurrences of x in its body P.
- For example, c is **free** in $isFriday \rightarrow Rainy(c)$, but **bound** in $\exists c \in Cities.\ isFriday \rightarrow Rainy(c)$.

sli.do 1275413

Let-bindings

- We will begin by looking at a simple type of binding: a **let-binding**
- The key idea is that we can give a name to a subexpression in a given continuation, so we don't need to repeat it
  - You can think of a let-binding as an **abbreviation**
- For example, the following expression would evaluate to 225

```
let x = (5 + 10) in x * x
```

- We can also **nest** let-bindings: for example the expression on the right would evaluate to 15

```
let x = 5 in
let y = 10 in
x + y
```

L_{Let} Abstract Syntax

L_{Let} Abstract Syntax

```
Integers n
Variables x, y, z
Operators ⊙  ::= + | - | *  | /
Values V, W   ::= n
Terms L, M, N ::= x | n
                | L ⊙ M
                | let x = M in N
```

- We will build each sublanguage off L_{Arith} to concentrate on the core details
- The main differences are **variables** (ranged over by x,y,z) and **let binders**
- Note that **variables are not values** (we should never need to evaluate a variable)

sli.do 1275413

Binding and Bound Occurrences in L_{Let}

- Much like with predicate logic formulae, **let** expressions act as a binder for a variable.

```
let x = (5 + 10) in x * x
```

Binding occurrence of x

Bound occurrences of x

- Formally, `let x = M in N` binds all **free occurrences** of x in the function body N.
- In the above example, the body of the let is `x * x`. Here, both occurrences of x are free, so become bound.

sli.do 1275413

Free Variables in L_{Let}, formally

- We can write out a recursive function fv(M) to get the free variables of an expression

x on its own is a free variable

`fv(x) = {x}`

Integer literals don't contain variables

`fv(n) = ∅`

`fv(let x = M in N) = fv(M) ∪ (fv(N) \ {x})`

`fv(M ⊙ N) = fv(M) ∪ fv(N)`

The free variables of a binary operator are the union of the free variables of its operands

Any occurrence of x would be **bound** in N, so we need to remove it from the set of N's free variables

sli.do 1275413

Name Shadowing

- Remember, a let-expression only binds **free** occurrences of the variable in the body. Consider the following:

```
let x = 5 in
let x = 10 in
x + x
```

- The body of the first let-binder is `let x = 10 in x + x`
  - There are **no free occurrences of** x – both occurrences of x are instead bound by the second let-binder
- Therefore, the first let-binder is redundant: we say that it has been **shadowed** by a more recent binder
- The expression would evaluate to 20

sli.do 1275413

Scope

- The **scope** of a variable is the collection of program locations in which occurrences of the variable refer to the same thing
  - (With some caveats – for example when we consider mutation later on)

```
let x = 5 in
let y = 10 in
x + y
```

Scope of x

Scope of y

sli.do 1275413

Scope

- The **scope** of a variable is the collection of program locations in which occurrences of the variable refer to the same thing
  - (With some caveats – for example when we consider mutation later on)

Scope of first let-bound x

```
let x = 5 in
x +
  (let x = 10 in
   x + x)
```

Scope of second let-bound x

Scope

- The **scope** of a variable is the collection of program locations in which occurrences of the variable refer to the same thing
  - (With some caveats – for example when we consider mutation later on)

Scope of y

```
let x =
  let y = 5 in y
in
x * x
```

Scope of x

sli.do 1275413

Let-bindings are **not** imperative variable assignment!

- Remember that we are working in a language **without** mutation at the moment – we can shadow bindings, but data **cannot change**

- Consider the following:

```
let x = 5 in
let y =
  let x = 100 in x - x
in
x + y
```

This occurrence of x refers to 5

Both occurrences of x here refer to 100

- This would evaluate to 5, **not** 105

sli.do 1275413

Substitution (Intuition)

- To define the operational semantics for L_{Let} we need to first define a **substitution** operation

$\boxed{M\ \{V\ /\ x\}}$ "Replace all free occurrences of x in term M with value V"

Note: substitution is also written different ways in the literature, e.g. `M[V / x]`, `M[x ↦ V]` and `M[x := V]`
If you're interested, [this talk by Guy Steele](#) gives an interesting discussion of the variety of meta-notation used in PL papers

- Examples:
  - `(x + 10) { 100 / x } = 100 + 10`
  - `(y + 10) { 100 / x } = y + 10`
  - `(let x = 5 in x + y) { 10 / y } = let x = 5 in x + 10`
  - `(let x = 5 in x + y) { 10 / x } = let x = 5 in x + y`

sli.do 1275413

Substitution (Formal Definition)

$$x\{V/y\} = \begin{cases} V & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

If x is the same variable as the one we're substituting for, replace it with the value V. Otherwise, leave it as it was.

$$n\{V/y\} = n$$

Substitution doesn't affect integer literals.

$$(\textbf{let}\ x = M\ \textbf{in}\ N)\{y/z\} = \begin{cases} \textbf{let}\ x = M\{y/z\}\ \textbf{in}\ N & \text{if } x = y \\ \textbf{let}\ x = M\{y/z\}\ \textbf{in}\ N\{y/z\} & \text{otherwise} \end{cases}$$

Substitution only affects **free** occurrences of x. If x = y, then the previous binding for y is **shadowed** and the body will contain no free occurrences of x

$$(M \odot N)\{V/x\} = (M\{V/x\}) \odot (N\{V/x\})$$

Propagate substitutions into the subexpressions of binary operators

sli.do 1275413

L_{Let} Operational Semantics

First evaluate M to a value V

Then, substitute V for x in the body N, and evaluate it to a value W

$$\frac{\boxed{M \Downarrow V} \qquad \boxed{N\{V/x\} \Downarrow W}}{\boxed{\mathtt{let}\ x = M\ \mathtt{in}\ N} \Downarrow \boxed{W}}$$

To evaluate a let-expression…

The result of the expression is W

- There is no rule for variables : trying to evaluate a free variable is an error.
- Note that we're choosing to evaluate M **before** substitution. We call this approach **eager**, or **call-by-value**.
  - There are other potential choices, for example substituting the expression **before** evaluating it (called **call-by-name**). Alas we won't have time to go into that in more detail.

sli.do 1275413

L_{Let} Reduction Example

$$\frac{M \Downarrow V \qquad N\{V/x\} \Downarrow W}{\mathtt{let}\ x = M\ \mathtt{in}\ N \Downarrow W}$$

$$\frac{}{\mathtt{let}\ x = 10 + 15\ \mathtt{in}\ (\mathtt{let}\ y = 100\ \mathtt{in}\ x + y) \Downarrow ?}$$

sli.do 1275413

L_{Let} Reduction Example

$$\frac{M \Downarrow V \qquad N\{V/x\} \Downarrow W}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow W}$$

$$\cfrac{\cfrac{\cfrac{}{10 \Downarrow 10} \qquad \cfrac{}{15 \Downarrow 15}}{10 + 15 \Downarrow 25}}{\mathbf{let}\ x = 10 + 15\ \mathbf{in}\ (\mathbf{let}\ y = 100\ \mathbf{in}\ x + y) \Downarrow 125}$$

sli.do 1275413

L_{Let} Reduction Example

$$\frac{M \Downarrow V \qquad N\{V/x\} \Downarrow W}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow W}$$

$$\dfrac{\dfrac{\overline{10 \Downarrow 10} \qquad \overline{15 \Downarrow 15}}{10 + 15 \Downarrow 25} \qquad \overline{\mathbf{let}\ y = 100\ \mathbf{in}\ 25 + y \Downarrow ?}}{\mathbf{let}\ x = 10 + 15\ \mathbf{in}\ (\mathbf{let}\ y = 100\ \mathbf{in}\ x + y) \Downarrow ?}$$

sli.do 1275413

$\mathrm{L}_{\mathrm{Let}}$ Reduction Example

$$\frac{M \Downarrow V \qquad N\{V / x\} \Downarrow W}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow W}$$

$$\dfrac{\dfrac{\dfrac{}{10 \Downarrow 10} \qquad \dfrac{}{15 \Downarrow 15}}{10 + 15 \Downarrow 25} \qquad \dfrac{\dfrac{}{100 \Downarrow 100} \qquad \dfrac{\dfrac{}{25 \Downarrow 25} \qquad \dfrac{}{100 \Downarrow 100}}{25 + 100 \Downarrow 125}}{\mathbf{let}\ y = 100\ \mathbf{in}\ 25 + y \Downarrow ?}}{\mathbf{let}\ x = 10 + 15\ \mathbf{in}\ (\mathbf{let}\ y = 100\ \mathbf{in}\ x + y) \Downarrow ?}$$

sli.do 1275413

$\mathrm{L_{Let}}$ Reduction Example

$$\frac{M \Downarrow V \qquad N\{V/x\} \Downarrow W}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow W}$$

$$\dfrac{\dfrac{}{10 \Downarrow 10} \qquad \dfrac{}{15 \Downarrow 15}}{10 + 15 \Downarrow 25} \qquad \dfrac{\dfrac{}{100 \Downarrow 100} \qquad \dfrac{\dfrac{}{25 \Downarrow 25} \qquad \dfrac{}{100 \Downarrow 100}}{25 + 100 \Downarrow 125}}{\mathbf{let}\ y = 100\ \mathbf{in}\ 25 + y \Downarrow 125}$$
$$\overline{\mathbf{let}\ x = 10 + 15\ \mathbf{in}\ (\mathbf{let}\ y = 100\ \mathbf{in}\ x + y) \Downarrow ?}$$

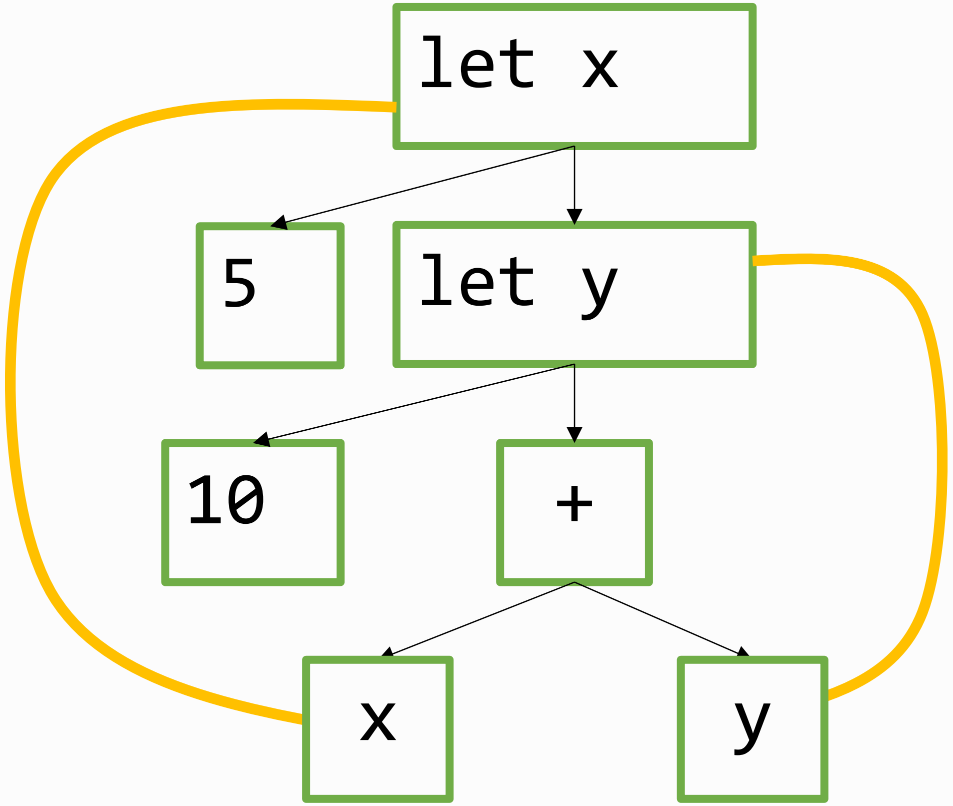
sli.do 1275413

L_{Let} Reduction Example

$$\frac{M \Downarrow V \qquad N\{V/x\} \Downarrow W}{\textbf{let}\ x = M\ \textbf{in}\ N \Downarrow W}$$

$$\dfrac{\dfrac{\dfrac{}{10 \Downarrow 10} \qquad \dfrac{}{15 \Downarrow 15}}{10 + 15 \Downarrow 25} \qquad \dfrac{\dfrac{}{100 \Downarrow 100} \qquad \dfrac{\dfrac{}{25 \Downarrow 25} \qquad \dfrac{}{100 \Downarrow 100}}{25 + 100 \Downarrow 125}}{\textbf{let}\ y = 100\ \textbf{in}\ 25 + y \Downarrow 125}}{\textbf{let}\ x = 10 + 15\ \textbf{in}\ (\textbf{let}\ y = 100\ \textbf{in}\ x + y) \Downarrow 125}$$

Binding Diagrams



- Sometimes it helps to visualise the binding structure for an expression
- On the left, we have a diagrammatic representation of the abstract syntax tree for the expression

```
let x = 5 in
let y = 10 in
x + y
```

- We can visualise the binding structure by drawing connecting lines between binding and bound occurrences of each variable

sli.do 1275413

α-equivalence

- It is useful to treat two expressions as the same, as long as their **binding structure** is the same. For example, we can equate the following expressions:

```
let x = 5 in
let y = 10 in
x + y
```

$\approx_\alpha$

```
let a = 5 in
let b = 10 in
a + b
```

- This is because we **consistently** rename **bound variables** x to a, and y to b.

sli.do 1275413

Non-examples of α-equivalence

`x`	$\not\approx_\alpha$	`y`	Only **bound** variables can be renamed – in this case, x and y are distinct **free** variables

`let x = 5 in`
`let y = 10 in`
`x + y`
	$\not\approx_\alpha$	`let a = 5 in`
		`let a = 10 in`
		`a + a`
			This example **changes the binding structure** since both occurrences of a refer to the second binder

`let x = 5 in`
`let y = 10 in`
`x + y`
	$\not\approx_\alpha$	`let x = 5 in`
		`x + y`
			The right expression is syntactically different – the second let-binder is removed

`let x = 5 in`
`let y = 10 in`
`x + y`
	$\not\approx_\alpha$	`let x = 5 in`
		`let y = 10 in`
		`y + x`
			Even though + is commutative, the structure has changed

sli.do 1275413

Determining α-equivalence by renaming

- We can determine whether two terms are α-equivalent by consistently renaming all bound variables, and then checking syntactic equality

```
let x = 5 in
let y = 10 in
x + y
```

Rename x to c, and y to d

```
let c = 5 in
let d = 10 in
c + d
```

= Expressions are equal

```
let a = 5 in
let b = 10 in
a + b
```

Rename a to c, and b to d

```
let c = 5 in
let d = 10 in
c + d
```

sli.do 1275413

Determining α-equivalence by comparing binding diagrams

- Alternatively, we can draw binding diagrams for both expressions

- The only place the diagrams are allowed to be different is the name of the let-binder, and the names of bound variables
  - In this case, the connecting lines must be identical

sli.do 1275413

Conclusion

- In this lecture we've seen:
  - Free and bound variables
  - Let-bindings, and their scoping
  - Substitution and operational semantics for L_{Let}
  - Abstract binding diagrams and α-equivalence

- See you in 10 minutes, when we'll talk about functions and recursion!