# Functions

What we here call 'variables' are variables in the logical sense—placeholders standing for unknown quantities and for which substitutions can be made. In the context of programming languages they are often called 'identifiers', because what we here refer to as locations are very often called 'variables' (because their contents may vary during execution and because it is common to use the name of a storage location without qualification to denote its contents). What we here call 'function abstractions' are also called *lambda abstractions* because of the notation $\lambda x.\, M$ introduced by Church in his lambda calculus.

**Expressions of the LFP language**

$$
\begin{aligned}
M ::= {} & n \mid b \mid \ell \mid M \; iop \; M \mid M \; bop \; M \\
& \mid \textbf{if } M \textbf{ then } M \textbf{ else } M \mid !M \mid M := M \\
& \mid \textbf{skip} \mid M \,;\, M \mid \textbf{while } M \textbf{ do } M \\
& \mid x \mid \lambda x.\, M \mid M \; M
\end{aligned}
$$

where
$x \in \mathbb{V}$, an infinite set of *variables*,
$n \in \mathbb{Z}$ (integers), $b \in \mathbb{B}$ (booleans), $\ell \in \mathbb{L}$ (locations),
$iop \in \mathbb{I}op$ (integer-valued binary operations), and $bop \in \mathbb{B}op$ (boolean-valued binary operations).

**Slide 37**

# Substitution and $\alpha$-conversion

When it comes to function application, the operational semantics of LFP will involve the syntactic operation $M[M'/x]$ of *substituting an expression $M'$ for all free occurrences of the variable $x$ in the expression $M$*. This operation involves several subtleties, illustrated on Slide 38, which arise from the fact that $M \mapsto \lambda x.\, M$ is a variable-binding operation. The

occurrence of $x$ next to $\lambda$ in $\lambda x.\, M$ is a *binding occurrence* of the variable $x$ whose *scope* is the whole syntax tree $M$; and in $\lambda x.\, M$ no occurrences of $x$ in $M$ are free for substitution by another expression (see example (ii) on Slide 38). The finite set of *free variables* of an expression is defined on Slide 39. The key clause is the last one—$x$ is not a free variable of $\lambda x.\, M$.

In fact we need the operation of *simultaneously* substituting expressions for a number of different free variables in an expression. Given a *substitution* $\sigma$, i.e. a finite partial function mapping variables to LFP expressions, $M[\sigma]$ will denote the LFP expression resulting from simultaneous substitution of each $x \in dom(\sigma)$ by the corresponding expression $\sigma(x)$. It is defined by induction on the structure of $M$ (simultaneously for all substitutions $\sigma$) in Figure 5 (cf. Stoughton 1988). Then we can take $M[M'/x]$ to be $M[\sigma]$ with $\sigma = \{x \mapsto M'\}$.

**Substitution examples**

$M[M'/x]$ — substitute $M'$ for all free occurrences of the variable $x$ in the expression $M$.

(i) $(\lambda x.\, x + y)[4/y]$ is $\lambda x.\, x + 4$.

(ii) $(\lambda x.\, x + y)[4/x]$ is $\lambda x.\, x + y$, not $\lambda x.\, 4 + y$, because $\lambda x.\, x + y$ contains no *free* occurrence of $x$.

(iii) $\lambda x.\, x + y$ is the same as (is $\alpha$-*convertible* with) $\lambda z.\, z + y$; and $(\lambda x.\, x + y)[x/y]$ is $\lambda z.\, z + x$, not $\lambda x.\, x + x$.

**Slide 38**

$fv(M)$ — **set of *free variables* of** $M$

$$fv(n) = fv(b) = fv(\ell) = fv(\mathbf{skip}) \stackrel{\text{def}}{=} \emptyset$$

$$fv(!M) \stackrel{\text{def}}{=} fv(M)$$

$$fv(M \ op \ M') = fv(M := M') = fv(M\ ;\ M') =$$
$$= fv(\mathbf{while}\ M\ \mathbf{do}\ M') = fv(M\ M') \stackrel{\text{def}}{=} fv(M) \cup fv(M')$$

$$fv(\mathbf{if}\ M\ \mathbf{then}\ M'\ \mathbf{else}\ M'') \stackrel{\text{def}}{=} fv(M) \cup fv(M') \cup fv(M'')$$

$$fv(x) \stackrel{\text{def}}{=} \{x\}$$

$$fv(\lambda x.\, M) \stackrel{\text{def}}{=} \{x' \in fv(M) \mid x' \neq x\}.$$

**Slide 39**

- $x[\sigma] \stackrel{\text{def}}{=} \begin{cases} \sigma(x) & \text{if } x \in dom(\sigma) \\ x & \text{otherwise.} \end{cases}$
- $n[\sigma] \stackrel{\text{def}}{=} n$. Similarly for $b$, $\ell$, and $\mathbf{skip}$.
- $(M\,M')[\sigma] \stackrel{\text{def}}{=} (M[\sigma])\,(M'[\sigma])$. Similarly for $M\ op\ M'$, $M := M'$, $M\ ;\ M'$, $\mathbf{while}\ M\ \mathbf{do}\ M'$, $!M$, and $\mathbf{if}\ M\ \mathbf{then}\ M'\ \mathbf{else}\ M''$.
- $(\lambda x.\, M)[\sigma] \stackrel{\text{def}}{=} \lambda x'.(M[\sigma[x \mapsto x']])$, where $x'$ is the first variable not in $fv(\sigma) \cup fv(M)$.

**Notes**

In the last clause of the definition:

- $\sigma[x \mapsto x']$ is the substitution mapping $x$ to $x'$ and otherwise acting like $\sigma$.
- $x'$ is first with respect to some fixed ordering of the set $\mathbb{V}$ of variables that we assume is given.
- $fv(\sigma) \stackrel{\text{def}}{=} \{y \mid \exists x \in dom(\sigma).\, y \in fv(\sigma(x))\}$ is the set of all free variables in the expressions being substituted by $\sigma$.
- Since $x' \notin fv(\sigma) \cup fv(M)$, the only occurrences of $x'$ in $M[\sigma[x \mapsto x']]$ that are 'captured' by $\lambda x'.(-)$ correspond to occurrences of $x$ in $M$ that were bound in $\lambda x.\, M$.

---

Figure 5: Definition of substitution

**$\alpha$-Conversion relation**

is inductively defined by the following axioms and rules:

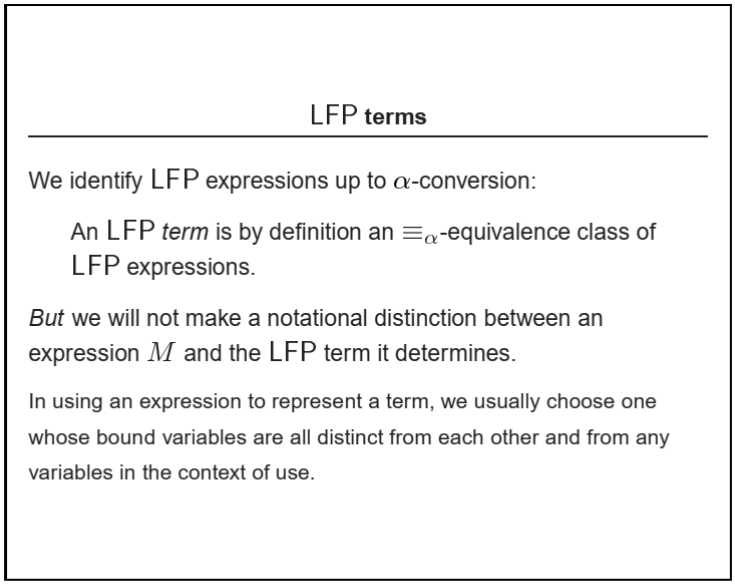
$$\lambda x.\, M \equiv_\alpha \lambda x'.(M[x'/x]) \qquad M \equiv_\alpha M$$

$$\frac{M \equiv_\alpha M'}{M' \equiv_\alpha M} \qquad \frac{M \equiv_\alpha M' \quad M' \equiv_\alpha M''}{M \equiv_\alpha M''}$$

$$\frac{M \equiv_\alpha M'}{\lambda x.\, M \equiv_\alpha \lambda x.\, M'} \qquad \frac{M_1 \equiv_\alpha M_1' \quad M_2 \equiv_\alpha M_2'}{M_1\, M_2 \equiv_\alpha M_1'\, M_2'}$$

plus rules like the last one for each of the other LFP expression-forming constructs.

**Slide 40**




**Slide 4**

Note how the last clause in Figure 5 avoids the problem of unwanted 'capture' of free variables in an expression being substituted, illustrated by example (iii) on Slide 38. It does so by '$\alpha$-converting' the bound variable. There is no problem with this from a semantical point of view, since in general we expect the meaning of a function abstraction to be independent of the name of the bound variable—$\lambda x.\, M$ and $\lambda x'.\, M[x'/x]$ should always mean the same thing. The equivalence relation $\equiv_\alpha$ of *$\alpha$-conversion* between LFP expressions is defined on Slide 40. In Section 2.1 we noted that the representation of syntax as parse trees rather than as strings of symbols is the proper level of abstraction when discussing programming language semantics. In fact when the language involves binding constructs one should take this a step further and use a representation of syntax that identifies $\alpha$-convertible expressions. It is possible to do this in a clever way that still allows expressions to be tree-like data structures through the use of de Bruijn's 'nameless terms', but at the expense of complicating the definition of substitution: the interested reader is referred to (Barendregt 1984, Appendix C). Here we will use brute force and quotient the set of expressions by the equivalence relation $\equiv_\alpha$: see Slide 41. The convention mentioned on that slide—not making any notational distinction between an expression $M$ and the LFP term it determines—is possible because the operations on syntax that we will employ all respect $\alpha$-conversion. For example, and as one might expect, it is the case that the operation of substitution respects $\equiv_\alpha$:

$$(M_1 \equiv_\alpha M_1' \;\&\; M_2 \equiv_\alpha M_2') \Rightarrow M_1[M_2/x] \equiv_\alpha M_1'[M_2'/x].$$

Similarly, the set of free variables of an expression is invariant with respect to $\equiv_\alpha$:

$$M \equiv_\alpha M' \quad \Rightarrow \quad fv(M) = fv(M').$$

## Call-by-name and call-by-value

We will give the structural operational semantics of LFP in terms of an inductively defined relation of evaluation whose form is shown on Slide 42. Compared with LC, the main novelty lies in the rules for evaluating function abstractions and function application. For function abstractions, we take configurations of the form $\langle \lambda x.\, M, s \rangle$ to be terminal. For function application, there are (at least) two natural strategies, depending upon whether or not an argument is evaluated before it is passed to the body of a function abstraction. These strategies are shown on Slide 43. Many pragmatic considerations to do with implementation influence which one to choose. The different strategies also radically alter the properties of evaluation and the ease with which one can reason about program properties—we shall see something of this below.

### LFP evaluation relation

takes the form:

$$\langle M,\, s \rangle \Downarrow \langle V,\, s' \rangle$$

where

- $M$ and $V$ are *closed* terms, i.e. $fv(M) = fv(V) = \emptyset$.
- $s$ and $s'$ are *states*, i.e. finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$.
- $V$ is a *value*, $V ::= n \mid b \mid \ell \mid \mathbf{skip} \mid \lambda x.\, M$.

**Slide 42**

**Call-by-name and call-by-value evaluation**

$$(\Downarrow_{\mathrm{cbn}}) \qquad \frac{\langle M_1,\, s\rangle \Downarrow \langle \lambda x.\, M_1',\, s'\rangle \quad \langle M_1'[M_2/x],\, s'\rangle \Downarrow \langle V,\, s''\rangle}{\langle M_1\ M_2,\, s\rangle \Downarrow \langle V,\, s''\rangle}$$

$$(\Downarrow_{\mathrm{cbv}}) \qquad \frac{\langle M_1,\, s\rangle \Downarrow \langle \lambda x.\, M_1',\, s'\rangle \quad \langle M_2,\, s'\rangle \Downarrow \langle V_2,\, s''\rangle \quad \langle M_1'[V_2/x],\, s''\rangle \Downarrow \langle V,\, s'''\rangle}{\langle M_1\ M_2,\, s\rangle \Downarrow \langle V,\, s'''\rangle}$$

**Slide 43**

$$(\Downarrow_{\mathrm{val}}) \qquad \langle V, s\rangle \Downarrow \langle V, s\rangle \quad (V \text{ a value})$$

$$(\Downarrow_{\mathrm{op}}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle M_1\ op\ M_2, s\rangle \Downarrow \langle c, s''\rangle}$$

where $c$ is the value of $n_1\ op\ n_2$ (for $op$ an integer or boolean operation)

$$(\Downarrow_{\mathrm{if1}}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle \mathbf{true}, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle V, s''\rangle}{\langle \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3, s\rangle \Downarrow \langle V, s''\rangle}$$

$$(\Downarrow_{\mathrm{if2}}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle \mathbf{false}, s'\rangle \quad \langle M_3, s'\rangle \Downarrow \langle V, s''\rangle}{\langle \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3, s\rangle \Downarrow \langle V, s''\rangle}$$

$$(\Downarrow_{!}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle \ell, s'\rangle}{\langle !M_1, s\rangle \Downarrow \langle n, s'\rangle} \quad \text{if } \ell \in dom(s') \ \&\ s'(\ell) = n$$

$$(\Downarrow_{:=}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle \ell, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle n, s''\rangle}{\langle M_1 := M_2, s\rangle \Downarrow \langle \mathbf{skip}, s''[\ell \mapsto n]\rangle}$$

$$(\Downarrow_{\mathrm{seq}}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}{\langle M_1\ ;\ M_2, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}$$

$$(\Downarrow_{\mathrm{wh1}}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle \mathbf{true}, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle \quad \langle \mathbf{while}\ M_1\ \mathbf{do}\ M_2, s''\rangle \Downarrow \langle \mathbf{skip}, s'''\rangle}{\langle \mathbf{while}\ M_1\ \mathbf{do}\ M_2, s\rangle \Downarrow \langle \mathbf{skip}, s'''\rangle}$$

$$(\Downarrow_{\mathrm{wh2}}) \qquad \frac{\langle M_1, s\rangle \Downarrow \langle \mathbf{false}, s'\rangle}{\langle \mathbf{while}\ M_1\ \mathbf{do}\ M_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}$$

plus either rule ($\Downarrow_{\mathrm{cbn}}$) or rule ($\Downarrow_{\mathrm{cbv}}$) on Slide 43.

Figure 6: Axioms and rules for LFP evaluation

**For LFP, $\Downarrow_{\mathrm{n}}$ and $\Downarrow_{\mathrm{v}}$ are incomparable**

Let

$$C_0 \stackrel{\text{def}}{=} \textbf{while true do skip}$$

$$C_1 \stackrel{\text{def}}{=} (\lambda x.\, \textbf{skip})\, C_0$$

$$C_2 \stackrel{\text{def}}{=} (\lambda x.\, \textbf{if } !\ell = 0 \textbf{ then skip else } C_0)\, (\ell := 0).$$

Then

$$\langle C_1, s\rangle \Downarrow_{\mathrm{n}} \langle \textbf{skip}, s\rangle \quad \text{(any } s\text{)}$$
$$\langle C_1, s\rangle \not\Downarrow_{\mathrm{v}}$$

$$\langle C_2, \{\ell \mapsto 1\}\rangle \not\Downarrow_{\mathrm{n}}$$
$$\langle C_2, \{\ell \mapsto 1\}\rangle \Downarrow_{\mathrm{v}} \langle \textbf{skip}, \{\ell \mapsto 0\}\rangle.$$

**Slide 44**

**Definition** The *call-by-name* (respectively *call-by-value*) *evaluation relation* for LFP terms is denoted $\Downarrow_{\mathrm{n}}$ (respectively $\Downarrow_{\mathrm{v}}$) and is inductively generated by the rule ($\Downarrow_{\mathrm{cbn}}$) (respectively ($\Downarrow_{\mathrm{cbv}}$)) on Slide 43 together with the axioms and rules in Figure 6.

The examples on Slide 44 exploit the fact that evaluation of LFP terms can cause state change to show that there is no implication either way between call-by-value convergence and call-by-name convergence. The following notation is used on the slide:

$$\langle M, s\rangle \not\Downarrow \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \text{there is no } \langle V, s'\rangle \text{ for which } \langle M, s\rangle \Downarrow \langle V, s'\rangle \text{ holds.}$$

We leave the verification of these examples as simple exercises.

## Static semantics

As things stand, there are many LFP terms that do not evaluate to anything because of *type* mis-matches. For example, although the application of an integer to a function, such as $2\ (\lambda x.\, x)$, is a legal expression, it is not really a meaningful one. We can weed out such things by assigning types to LFP terms using a relation of the kind shown on Slide 45. The intended meaning of $\Gamma \vdash M : \tau$ is:

> "If the variable $x$ has type $\Gamma(x)$ for each $x \in dom(\Gamma)$, then the term $M$ has type $\tau$."

We capture this intention through an inductive definition of the relation that follows the structure of the term $M$. The rules for function abstraction and application are shown on Slide 46 and the other axioms and rules in Figure 7. Note that these rules apply to *terms*, i.e. to expressions up to $\alpha$-conversion. Thus

$$\dfrac{\dfrac{}{\{x' \mapsto \tau', x \mapsto \tau\} \vdash x : \tau}\ (:_{\mathrm{var}})}{\{x' \mapsto \tau'\} \vdash \lambda x'.\, x' : \tau \to \tau}\ (:_{\mathrm{fn}})$$

is a valid application of the rules, because $\lambda x'.\, x'$is the same term as $\lambda x.\, x$. In using the rules from the bottom up to deduce a type for a term $M$, it is as well to use a representative expression for $M$ that has all its bound variables distinct from each other and from the variables in the domain of definition of the type environment. So for example $\emptyset \vdash \lambda x.\, \lambda x.\, x : \tau \to (\tau' \to \tau')$ holds, but it is probably easier to deduce this using the $\alpha$-equivalent expression $\lambda x.\, \lambda x'.\, x'$.

**Typing relation**

takes the form $\boxed{\Gamma \vdash M : \tau}$ where:

- $\tau$ is a *type* ::=
  - $int$ integers
  - $\mid$ $bool$ booleans
  - $\mid$ $loc$ location
  - $\mid$ $cmd$ commands
  - $\mid$ $\tau \to \tau$ functions.
- $\Gamma$ is a *type environment*, i.e. a finite partial function mapping variables to types.
- $M$ is an LFP term.

**Slide 45**

$(:_{\text{var}})$ $\quad \Gamma \vdash x : \tau \quad$ if $x \in dom(\Gamma)$ & $\Gamma(x) = \tau$

$(:_{\text{int}})$ $\quad \Gamma \vdash n : int \quad (n \in \mathbb{Z})$

$(:_{\text{bool}})$ $\quad \Gamma \vdash b : bool \quad (b \in \mathbb{B})$

$(:_{\text{loc}})$ $\quad \Gamma \vdash \ell : loc \quad (\ell \in \mathbb{L})$

$(:_{\text{iop}})$ $\quad \dfrac{\Gamma \vdash M_1 : int \quad \Gamma \vdash M_2 : int}{\Gamma \vdash M_1 \; iop \; M_2 : int}$

$(:_{\text{bop}})$ $\quad \dfrac{\Gamma \vdash M_1 : int \quad \Gamma \vdash M_2 : int}{\Gamma \vdash M_1 \; bop \; M_2 : bool}$

$(:_{\text{if}})$ $\quad \dfrac{\Gamma \vdash M_1 : bool \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M_3 : \tau}$

$(:_{\text{get}})$ $\quad \dfrac{\Gamma \vdash M : loc}{\Gamma \vdash !M : int}$

$(:_{\text{set}})$ $\quad \dfrac{\Gamma \vdash M_1 : loc \quad \Gamma \vdash M_2 : int}{\Gamma \vdash M_1 := M_2 : cmd}$

$(:_{\text{skip}})$ $\quad \Gamma \vdash \textbf{skip} : cmd$

$(:_{\text{seq}})$ $\quad \dfrac{\Gamma \vdash M_1 : cmd \quad \Gamma \vdash M_2 : cmd}{\Gamma \vdash M_1 \, ; M_2 : cmd}$

$(:_{\text{whl}})$ $\quad \dfrac{\Gamma \vdash M_1 : bool \quad \Gamma \vdash M_2 : cmd}{\Gamma \vdash \textbf{while } M_1 \textbf{ do } M_2 : cmd}$

plus rules $(:_{\text{fn}})$ and $(:_{\text{app}})$ on Slide 46.

Figure 7: Axioms and rules for LFP typing

**Typing rules for function abstraction and application**

$(:_{\text{fn}})$ $\quad \dfrac{\Gamma[x \mapsto \tau] \vdash M : \tau'}{\Gamma \vdash \lambda x.\, M : \tau \to \tau'}$ if $x \notin dom(\Gamma)$

$(:_{\text{app}})$ $\quad \dfrac{\Gamma \vdash M_1 : \tau \to \tau' \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 \, M_2 : \tau'}$

In rule $(:_{\text{fn}})$, $\Gamma[x \mapsto \tau]$ denotes the type environment mapping $x$ to $\tau$ and otherwise acting like $\Gamma$.

**Slide 46**

**Definition 5.3.1 (Typeable closed terms).** Given a closed LFP term $M$ (i.e. one with no free variables), we say $M$ *has type* $\tau$ and write

$$M : \tau$$

if $\emptyset \vdash M : \tau$ is derivable from the axioms and rules in Figure 7 (and Slide 46).

Note that an LFP term may have several different types—for example $\lambda x.\, x$ has type $\tau \to \tau$ for any $\tau$. This is because we have not built any explicit type information into the syntax of expressions—an explicitly typed function abstraction would tag its bound variable with a type: $(\lambda x : \tau.\, M)$. For LFP, there is an algorithm which, given $\Gamma$ and $M$, decides whether or not there exists a type $\tau$ satisfying $\Gamma \vdash M : \tau$. This is why this section is entitled *static* semantics: type checking is decidable and hence can be carried out at compile-time rather than at run-time. However, we will not pursue this topic of *type checking* here—see the Part II course on **Types**. Rather, we wish to indicate how types can be used to predict some aspects of the dynamic behaviour of terms. Slide 47 gives two examples of this. Both properties rely on the following substitution property of the typing relation.

**Lemma 5.3.2.** *If* $\Gamma \vdash M : \tau$ *and* $\Gamma[x \mapsto \tau] \vdash M' : \tau'$ *with* $x \notin dom(\Gamma)$, *then* $\Gamma \vdash M'[M/x] : \tau'$.

This can be proved by induction on the structure of $M'$; we omit the details.

## Local recursive definitions

In this section we consider the operational semantics of various kinds of *local declaration*, concentrating on *lexically scoped* constructs, i.e. ones whose scopes can be decided purely from the syntax of the language, at compile time. The designers of Algol 60 (Naur and Woodger (editors) 1963) defined the concept of locality for program blocks in their language as follows (quoted from Tennent 1991, page 84).

> "Any identifier occurring within a block may through a suitable declaration be specified to be local to the block in question. This means (a) that that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block."

The modern view (initiated by Landin 1966) is that for lexically scoped constructs, such matters can be made mathematically precise via the notions of *bound variable*, *substitution* and *α-conversion* from the lambda calculus For example, function abstraction and application in LFP can be combined to give *local definitions*, as shown on Slide 48.

**Local definitions in LFP**

$$\boxed{\textbf{let } x = M_1 \textbf{ in } M_2 \quad \stackrel{\text{def}}{=} \quad (\lambda x.\, M_2)\, M_1}$$

*Derived typing rule*:

$$\frac{\Gamma \vdash M_1 : \tau \qquad \Gamma[x \mapsto \tau] \vdash M_2 : \tau'}{\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau'}$$

*Derived evaluation rule* (call-by-name):

$$\frac{\langle M_2[M_1/x], s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle}{\langle \textbf{let } x = M_1 \textbf{ in } M_2, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle}$$

**Slide 48**

Note that $fv(\textbf{let } x = M_1 \textbf{ in } M_2) = fv(M_1) \cup \{x' \in fv(M_2) \mid x' \neq x\}$ and that free occurrences of $x$ in $M_2$ become bound in $\textbf{let } x = M_1 \textbf{ in } M_2$. Slide 49 illustrates how locality is enforced via $\alpha$-conversion.

**Remark 5.4(1)** Given the definition of $\textbf{let } x = M_1 \textbf{ in } M_2$ on Slide 48, the typing and evaluation rules given on the slide are *derivable* from the rules for call-by-name LFP in the sense that

- if $\Gamma \vdash M_1 : \tau$ and $\Gamma[x \mapsto \tau] \vdash M_2 : \tau'$ are derivable from Figure 7, then so is $\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau'$;
- if $\langle M_2[M_1/x], s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle$, then $\langle \textbf{let } x = M_1 \textbf{ in } M_2, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle$.

Remember that we only defined evaluation for *closed* LFP terms. So in the evaluation rule $M_1$ is a closed term and $M_2$ contains at most $x$ free.

(ii) For *call-by-value* evaluation of a local definition, rule ($\Downarrow_{\mathrm{cbv}}$) on Slide 43 means that we first compute the value of $M_1$ (if any) and use that as the local definition of $x$ in evaluating $M_2$. So

- if $\langle M_1, s\rangle \Downarrow_{\mathrm{v}} \langle V_1, s'\rangle$ and $\langle M_2[V_1/x], s'\rangle \Downarrow_{\mathrm{v}} \langle V_2, s''\rangle$, then $\langle \textbf{let } x = M_1 \textbf{ in } M_2, s\rangle \Downarrow_{\mathrm{v}} \langle V_2, s''\rangle$.

**Locality via $\alpha$-conversion**

Because we identify LFP expressions up to $\alpha$-conversion, the particular name of a bound variable is immaterial:

$\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ and $\mathbf{let}\ x' = M_1\ \mathbf{in}\ M_2[x'/x]$ represent the *same* LFP term.

Moreover, up to $\alpha$-conversion, a bound variable is always distinct from any variable in the surrounding context. For example:

$$(\mathbf{let}\ x = 1\ \mathbf{in}\ x + x) * x \equiv_\alpha (\mathbf{let}\ x' = 1\ \mathbf{in}\ x' + x') * x.$$

**Slide 49**

Note that the definition $x = M_1$ that occurs in $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ is a 'direct' definition—$x$ is being declared as a local abbreviation for $M_1$ in $M_2$. By contrast, a *recursive* definition such as

(34)
$$f = \lambda x.\ \mathbf{if}\ x < 0\ \mathbf{then}\ f\,x\ \mathbf{else\ if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * f(x-1)$$

in which the variable $f$ occurs (freely) in the right-hand side, has an altogether more complicated intended meaning: $f$ is supposed to be some data (a function in this case) that satisfies the equation (34). What does this really mean? To give a denotational semantics (cf. Slide 3) requires one to model data as suitable mathematical structures for which 'fixed point equations' such as (34) always have solutions; and to do this in full generality requires some fascinating mathematics that, alas, is not part of this course. The operational reading of (34) is the *unfolding rule*:

> "During evaluation of an expression in the scope of the definition (34), whenever a use of $f$ is encountered, use the right-hand side of the equation (thereby possibly introducing further uses of $f$) and continue evaluating."

In order to formulate this precisely, let us introduce an extension LFP with local recursive definitions, called $\mathsf{LFP}^+$. The expressions of $\mathsf{LFP}^+$ are given by the grammar for LFP (Slide 37) augmented by the **letrec** construct shown on Slide 50. Free occurrences of $x$

in $M_1$ and in $M_2$ become bound in $\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2$ and the extension to $\mathsf{LFP}^+$ of the definition of substitution given in Figure 5 is:

$$(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2)[\sigma] \quad \stackrel{\text{def}}{=} \quad \mathbf{letrec}\ x' = M_1[\sigma[x \mapsto x']]\ \mathbf{in}\ M_2[\sigma[x \mapsto x']]$$

where $x'$ is the first variable not in $fv(\sigma) \cup fv(M_1) \cup fv(M_2)$. We continue with the convention on Slide 41 and refer to $\equiv_\alpha$-equivalence classes as $\mathsf{LFP}^+$ *terms*. Of course the $\alpha$-conversion relation has to be suitably extended to cope with **letrec** expressions, by adding the axiom

$$(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2) \equiv_\alpha (\mathbf{letrec}\ x' = M_1[x'/x]\ \mathbf{in}\ M_2[x'/x])$$

and the rule

$$\frac{M_1 \equiv_\alpha M_1' \quad M_2 \equiv_\alpha M_2'}{(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2) \equiv_\alpha (\mathbf{letrec}\ x = M_1'\ \mathbf{in}\ M_2')}\ .$$

$\mathsf{LFP}^+$ **= LFP + local recursive definitions**

*Expressions*:

$$M ::= \cdots \mid \boxed{\mathbf{letrec}\ x = M\ \mathbf{in}\ M}$$

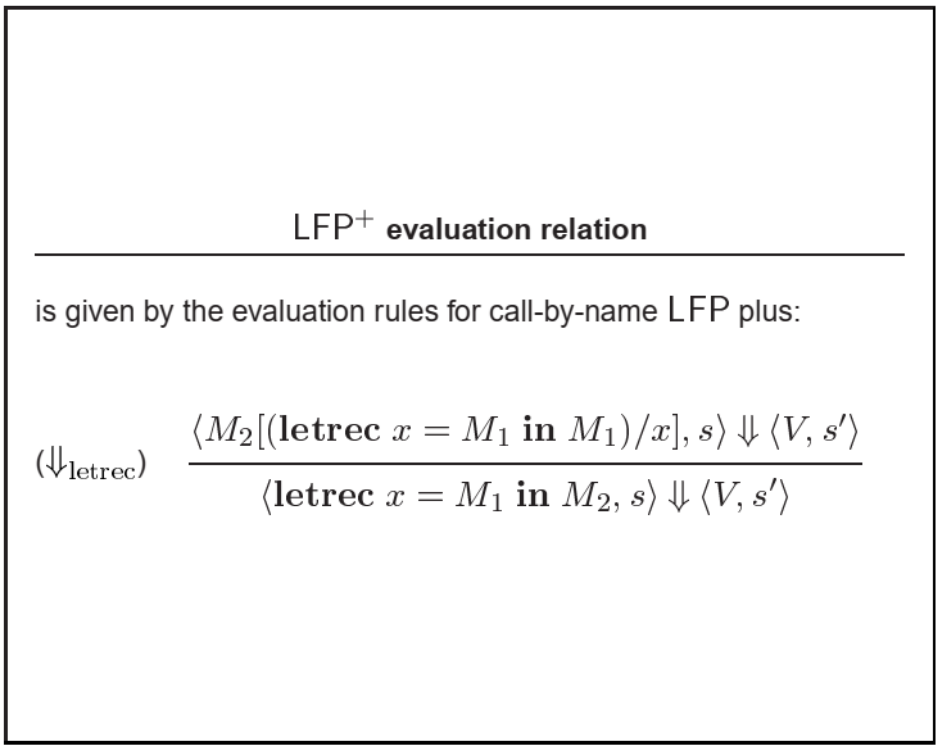
*Free variables*:

$$fv(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2) \stackrel{\text{def}}{=} \{x' \in fv(M_1) \cup fv(M_2) \mid x' \neq x\}$$

*Typing*:

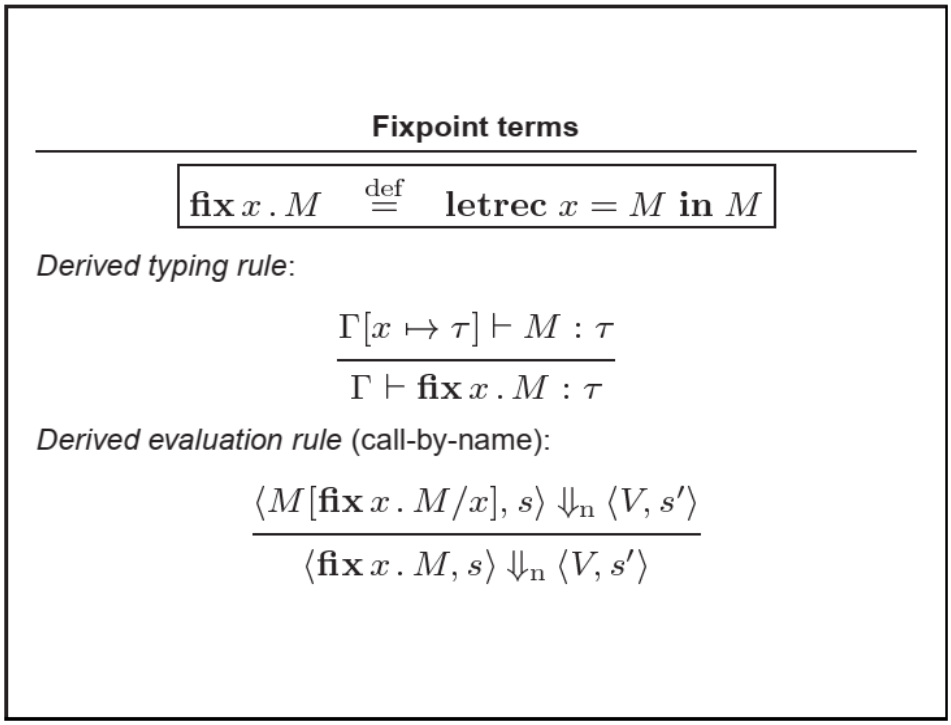
$$(:_{\text{letrec}}) \qquad \frac{\Gamma[x \mapsto \tau] \vdash M_1 : \tau \quad \Gamma[x \mapsto \tau] \vdash M_2 : \tau'}{\Gamma \vdash \mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2 : \tau'}$$

**Slide 50**




**Slide 51**

The static semantics of $\mathsf{LFP}^+$ is given by the typing axioms and rules for LFP (Figure 7) together with the rule $(:_{\mathrm{letrec}})$ on Slide 50. The $\mathsf{LFP}^+$ evaluation relation is inductively defined by the axioms and rules for call-by-name LFP augmented by the rule $(\Downarrow_{\mathrm{letrec}})$ on Slide 51; we will continue to denote it by $\Downarrow_{\mathrm{n}}$. Note the similarity with the call-by-name evaluation of non-recursive **let**-expressions (Slide 48). The difference is that when $M_1$ is substituted for $x$ in $M_2$, it is surrounded by the recursive definition of $x$.




**Slide 52**

Slide 52 shows the specialisation of the **letrec** construct to yield *fixpoint* terms. The typing and evaluation properties stated on the slide are direct consequences of the rules ($:_{\mathrm{letrec}}$) and ($\Downarrow_{\mathrm{letrec}}$). We make use of such terms in the following example.

**Example 5.4.2.**

(35) $\langle \mathbf{letrec}\, f = (\lambda x.\, \mathbf{if}\, x = 0\, \mathbf{then}\, 1\, \mathbf{else}\, x * f(x-1))\, \mathbf{in}\, f\, 1, s\rangle \Downarrow_{\mathrm{n}} \langle 1, s\rangle$.

*Proof.* Define

$$F \stackrel{\text{def}}{=} \mathbf{fix}\, f\,.\, \lambda x.\, M \stackrel{\text{def}}{=} \mathbf{letrec}\, f = \lambda x.\, M\, \mathbf{in}\, \lambda x.\, M$$

where

$$M \stackrel{\text{def}}{=} \mathbf{if}\, x = 0\, \mathbf{then}\, 1\, \mathbf{else}\, x * f(x-1).$$

For any closed term $N$ and value $V$ we have:

$$\dfrac{\dfrac{\dfrac{}{(\lambda x.\, M)[F/f] \Downarrow \lambda x.\, M[F/f]}\,(\Downarrow_{\mathrm{val}})}{F \Downarrow \lambda x.\, M[F/f]}\,(\Downarrow_{\mathrm{letrec}}) \qquad \begin{array}{c}\vdots\\ M[F/f][N/x] \Downarrow V\end{array}}{F\, N \Downarrow V}\,(\Downarrow_{\mathrm{cbn}})$$
$$\dfrac{}{\mathbf{letrec}\, f = \lambda x.\, M\, \mathbf{in}\, f\, N \Downarrow V}\,(\Downarrow_{\mathrm{letrec}})$$

where we have suppressed mention of the state part of configurations because it plays no part in this proof. Taking $N = V = 1$, we see that to prove (35), it suffices to prove $M[F/f][1/x] \Downarrow_{\mathrm{n}} 1$. But since $M[F/f][1/x] = \mathbf{if}\ 1 = 0\ \mathbf{then}\ 1\ \mathbf{else}\ 1 * F(1-1)$, for this it clearly suffices to prove that $F(1-1) \Downarrow_{\mathrm{n}} 1$. Taking $N = 1-1$ and $V = 1$ in the proof fragment shown above, we have that $F(1-1) \Downarrow_{\mathrm{n}} 1$ if $M[F/f][1-1/x] \Downarrow_{\mathrm{n}} 1$. But $M[F/f][1-1/x] = \mathbf{if}\ (1-1) = 0\ \mathbf{then}\ 1\ \mathbf{else}\ (1-1) * F((1-1)-1)$ and:

$$\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1}\,(\Downarrow_{\mathrm{val}}) \quad \dfrac{}{1 \Downarrow 1}\,(\Downarrow_{\mathrm{val}})}{1-1 \Downarrow 0}\,(\Downarrow_{\mathrm{op}}) \quad \dfrac{}{0 \Downarrow 0}\,(\Downarrow_{\mathrm{val}})}{(1-1)=0 \Downarrow \mathbf{true}}\,(\Downarrow_{\mathrm{op}}) \quad \dfrac{}{1 \Downarrow 1}\,(\Downarrow_{\mathrm{val}})}{\mathbf{if}\ (1-1)=0\ \mathbf{then}\ 1\ \mathbf{else}\ (1-1) * F((1-1)-1) \Downarrow 1}\,(\Downarrow_{\mathrm{if1}}).$$

## Exercises

**Exercise .1.** Consider the following LFP term for testing equality of location names in call-by-value LFP, where '$\mathbf{let}\ x = -\ \mathbf{in}\ -$' is as on Slide 48 and '**andthen**' is as in Remark 5.3.3.

$$\begin{aligned} eq \stackrel{\mathrm{def}}{=}\ & \lambda x_1.\, \lambda x_2.\, \mathbf{let}\ x = !x_1\ \mathbf{in} \\ & \quad x_1 := !x_2 + 1\ \mathbf{andthen} \\ & \quad \mathbf{if}\ !x_1 = !x_2\ \mathbf{then}\ (x_1 := x\ \mathbf{andthen}\ \mathbf{true}) \\ & \qquad \mathbf{else}\ (x_1 := x\ \mathbf{andthen}\ \mathbf{false}) \end{aligned}$$

Show that

$$\emptyset \vdash eq : loc \to (loc \to bool)$$

and that for all states $s$ and all $\ell, \ell' \in dom(s)$

$$\langle eq\, \ell\, \ell', s\rangle \Downarrow_{\mathrm{v}} \langle b, s\rangle \quad \text{where } b = \begin{cases} \mathbf{true} & \text{if } \ell = \ell' \\ \mathbf{false} & \text{if } \ell \neq \ell'. \end{cases}$$

**Exercise .2.** What is wrong with the following suggestion?

"The rule ($\Downarrow_{\mathrm{letrec}}$) on Slide 51 can be simplified to

$$\frac{\langle M_2[(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ x)/x], s\rangle \Downarrow \langle V, s'\rangle}{\langle \mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2, s\rangle \Downarrow \langle V, s'\rangle}$$

because in the body of the **letrec**-expression, $x$ is defined to be $M_1$ so we can use $\mathbf{letrec}\ x = M_1\ \mathbf{in}\ x$ instead of $\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_1$."

[Hint: consider $\mathbf{letrec}\ x = 0\ \mathbf{in}\ x$.]