9. Languages with Contexts II: An Applicative Language

Applicative Language

- Language without variables
  - *All identifiers are constant.*
  - *Attributes received at point of definition.*
- No assignment statement.
- No *Store* domain.
- Environment holds all identifier attributes.

Examples: arithmetic, pure LISP.

Pure LISP

See Figure 7.5

- Program = expression.
- Result = denotable value.
- Functions, lists, atoms.

  Denotable-value = (*(Denotable-value* $\rightarrow$ *Denotable-value)* + *Denotable-value** + *Atom* + *Error*)$_\perp$.

- Static scoping used

  Function body evaluated in context active at point of definition (not at point of use).

Language allows self-applicative behavior!

Scoping Rules

LET F = a_0 IN
    LET F = LAMBDA (Z) F CONS Z IN
        LET Z = a_1 IN
            F(Z CONS NL)

- Context of phrase solely determined by textual position.
- Occurence of F in function body refers to a_0!
- Simplification (see Figure 7.6)

  $\Rightarrow$ *(LAMBDA (Z)* a_0 *CONS Z) (*a_1 *CONS NIL)*
  $\Rightarrow$ *(*a_0 *CONS (*a_1 *CONS NIL))*

Dynamic Scoping

- Contex of phrase determined by place(s) where its value is required.
- Example: macro definition and invocation.
  - LET I=E *binds* I *to* *text* E.
  - *Invocation provides context for evaluation of* E.
- Semantics of abstraction and application:

E[[LAMBDA (I) E]] =
  $\lambda e.$ in*Function*(
    $\lambda e'.\lambda d.$ **E**[[E]](*updateenv* [[I]] d e'))
E[[E_1 E_2]] =
  $\lambda e.$ let x = (**E**[[E_1]]e) in
    cases x of
      is*Function*(f) $\rightarrow$ (f e (**E**[[E_2]]e))
      [] otherwise $\rightarrow$ in*Error*()
    end

Statically scoped languages are difficult to understand.

Example

LET X $=$ a_0 IN
    (LET Y $=$ X CONS NIL IN
        (LET X $=$ X CONS Y IN Y))
$\Rightarrow$ [X $\leftarrow$ a_0]
    LET Y $=$ X CONS NIL IN
        (LET X $=$ X CONS Y IN Y)
$\Rightarrow$ [X $\leftarrow$ a_0] [Y $\leftarrow$ X CONS NIL]
    LET X $=$ X CONS Y IN Y
$\Rightarrow$ [X $\leftarrow$ a_0] [Y $\leftarrow$ X CONS NIL] [X $\leftarrow$ X CONS Y]
    Y
$\Rightarrow$ [X $\leftarrow$ a_0] [Y $\leftarrow$ X CONS NIL] [X $\leftarrow$ X CONS Y]
    X CONS NIL
$\Rightarrow$ [X $\leftarrow$ a_0] [Y $\leftarrow$ X CONS NIL] [X $\leftarrow$ X CONS Y]
    (X CONS Y) CONS NIL
$\Rightarrow$ [X $\leftarrow$ a_0] [Y $\leftarrow$ X CONS NIL] [X $\leftarrow$ X CONS Y]
    (X CONS (X CONS Y)) CONS NIL
$\Rightarrow$ ...

Self-Application

- Untyped applicative language.
- LAMBDA expression can accept itself as argument.
- LET X = LAMBDA (X) (X X) IN (X X)

E[[LET X = LAMBDA (X) (X X) IN (X X)]]e_0
= **E**[[(X X)]]e_1 where
e_1 = (*updateenv* [[X]]
(**E**[[LAMBDA (X) (X X)]]e_0) e_0)
= **E**[[LAMBDA (X) (X X)]]e_0 (**E**[[X]]e_1)
= (λd.**E**[[(X X)]](*updateenv* [[X]]
d e_0))(**E**[[X]]e_1)
= **E**[[(X X)]](*updateenv* [[X]]
(**E**[[LAMBDA (X) (X X)]]e_0) e_0)
= **E**[[(X X)]]e_1
= ...

Circular derivation produced!

Self-Application

- Simplification on semantic expressions is not guarantueed to terminate.
  - *Some meaning exists in Denotable-value.*
  - *Which meaning is unclear.*
  - *Notation for representation meanings has shortcomings.*
  - *Inherent to all notations for functions.*
- Circular derivation produced without recursive definition.
  - *Recursion* $f = \alpha(f)$.
  - *Simulation* $h(g) = \alpha(g(g)), f = h(h)$.

    $f(p) = \lambda x.$ *if* $x = 0$ *then 1 else*
    $x * ((pp(p))(x - 1))$
    fac $= f(f)$
- Recursiveness in *Denotable-value*.

Problem solved by fixpoint theory of recursive domain definitions.

Recursive Definitions

- Mechanism for recursive LAMBDA forms
- Abstract syntax

  E ::= ... | LETREC I = E_1 IN E_2
  | IFNULL E_1 THEN E_2 ELSE E_3

- IFNULL is conditional on lists

  E[[IFNULL E_1 THEN E_2 ELSE E_3]] =
  λe. let $x = ($**E**[[E_1]]$e)$ in
  cases x of
  is*List*$(t) \rightarrow ((null\ t) \rightarrow$
  (**E**[[E_2]]e) [] (**E**[[E_3]]e))
  otherwise $\rightarrow$ in*Error*()
  end

Recursive Definitions

- Occurences of I in LETREC refer to the I being declared.

  E[[LETREC I = E_1 IN E_2]] = λe. **E**[[E_2]]e'
  where e' = *updateenv* [[I]] (**E**[[E_1]]e')e
  E[[LETREC I = E_1 IN E_2]] = λe. **E**[[E_2]]
  (*fix*($\lambda e'$. *updateenv* [[I]] (**E**[[E_1]]e')e))

- Example (see Figure 7.8)

  LETREC F =
  LAMBDA (X)
  IFNULL X THEN NIL
  ELSE a_0 CONS F(TAIL X)
  IN F(a_1 CONS a_2 CONS NIL)
  $\Rightarrow$ (a_0 CONS a_0 CONS NIL)

Fixed Point Semantics

- Functional G: *Environment* $\rightarrow$ *Environment*

$$
\begin{aligned}
G^0 &= \lambda i.\bot \\
G^1 &= \textit{updateenv}\ [[\mathsf{I}]]\ (\mathbf{E}[[E_1]](G^0))\ e \\
&= \textit{updateenv}\ [[\mathsf{I}]]\ (\mathbf{E}[[E_1]](\lambda i.\bot))\ e \\
G^2 &= \textit{updateenv}\ [[\mathsf{I}]]\ (\mathbf{E}[[E_1]](G^1))\ e \\
&= \textit{updateenv}\ [[\mathsf{I}]]\ (\mathbf{E}[[E_1]] \\
&\qquad (\textit{updateenv}\ [[\mathsf{I}]]\ (\mathbf{E}[[E_1]](G^1))\ e))\ e \\
&\ldots \\
G^{i+1} &= \textit{updateenv}\ [[\mathsf{I}]]\ (\mathbf{E}[[E_1]](G^i))\ e
\end{aligned}
$$

- Environment G^{i+1} can handle i recursive references to [[I]] in [[E_1]].
- Limit G can handle unlimited number of recursive references.

Not necessary to refer to theory to define and use recursive environments!

Substitution Principles

- **E**[[LET I = E_1 IN E_2]] = **E**[[[E_1/I]E_2]]

  LET X = a_0 IN
  LET Y = X CONS NIL IN
  (HEAD Y) CONS X CONS NIL
  ⇒ LET Y = a_0 CONS NIL IN
  (HEAD Y) CONS a_0 CONS NIL
  ⇒ (HEAD (a_0 CONS NIL)) CONS a_0 CONS NIL
  ⇒ a_0 CONS a_0 CONS NIL

- **E**[[LETREC I = E_1 IN E_2]] = ?
  - Substitute [[E_1]] for [[I]] in [[E_2]].
  - Substitute [[E_1]] for [[I]] in resulting expression.
  - Continue until occurences of [[I]] eliminated.
  - Number of substitutions is unbounded!
  - Substitute occurences only when required for further simplification!

Computation is substitution; in implementation environment becomes run-time structure.