Languages with Contexts II: An Applicative Language

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

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 is evaluated in the 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 <u>text</u> E.
  - Invocation provides context for evaluation of E.
- Semantics of abstraction and application:

$$
\begin{array}{l}
\textbf{E}[[\text{LAMBDA (I) E}]] = \\
\quad \lambda e.\ \text{in}\mathit{Function}(\\
\quad\quad \lambda e'.\lambda d.\ \textbf{E}[[\text{E}]](\mathit{updateenv}\ [[\text{I}]]\ d\ e')) \\
\textbf{E}[[\text{E}_1\ \text{E}_2]] = \\
\quad \lambda e.\ \text{let } x = (\textbf{E}[[\text{E}_1]]e) \text{ in} \\
\quad\quad \text{cases } x \text{ of} \\
\quad\quad\quad \text{is}\mathit{Function}(f) \rightarrow (f\ e\ (\textbf{E}[[E_2]]e)) \\
\quad\quad\quad [] \text{ otherwise} \rightarrow \text{in}\mathit{Error}() \\
\quad\quad \text{end}
\end{array}
$$

Statically scoped languages are difficult to understand.

Example

$$
\begin{array}{l}
\text{LET X} = \text{a}_0 \text{ IN} \\
\quad\quad \text{(LET Y} = \text{X CONS NIL IN} \\
\quad\quad\quad\quad \text{(LET X} = \text{X CONS Y IN Y))} \\
\Rightarrow [\text{X} \leftarrow \text{a}_0] \\
\quad\quad \text{LET Y} = \text{X CONS NIL IN} \\
\quad\quad\quad\quad \text{(LET X} = \text{X CONS Y IN Y)} \\
\Rightarrow [\text{X} \leftarrow \text{a}_0]\ [\text{Y} \leftarrow \text{X CONS NIL}] \\
\quad\quad \text{LET X} = \text{X CONS Y IN Y} \\
\Rightarrow [\text{X} \leftarrow \text{a}_0]\ [\text{Y} \leftarrow \text{X CONS NIL}]\ [\text{X} \leftarrow \text{X CONS Y}] \\
\quad\quad \text{Y} \\
\Rightarrow [\text{X} \leftarrow \text{a}_0]\ [\text{Y} \leftarrow \text{X CONS NIL}]\ [\text{X} \leftarrow \text{X CONS Y}] \\
\quad\quad \text{X CONS NIL} \\
\Rightarrow [\text{X} \leftarrow \text{a}_0]\ [\text{Y} \leftarrow \text{X CONS NIL}]\ [\text{X} \leftarrow \text{X CONS Y}] \\
\quad\quad \text{(X CONS Y) CONS NIL} \\
\Rightarrow [\text{X} \leftarrow \text{a}_0]\ [\text{Y} \leftarrow \text{X CONS NIL}]\ [\text{X} \leftarrow \text{X CONS Y}] \\
\quad\quad \text{(X CONS (X CONS Y)) CONS NIL} \\
\Rightarrow \ldots
\end{array}
$$

Self-Application

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

$$
\begin{aligned}
&\mathbf{E}[[\text{LET X = LAMBDA (X) (X X) IN (X X)}]]e_0\\
&= \mathbf{E}[[\text{(X X)}]]e_1 \text{ where}\\
&\quad\quad e_1 = (\textit{updateenv}\ [[\text{X}]]\\
&\quad\quad\quad\quad (\mathbf{E}[[\text{LAMBDA (X) (X X)}]]e_0)\ e_0)\\
&= \mathbf{E}[[\text{LAMBDA (X) (X X)}]]e_0\ (\mathbf{E}[[\text{X}]]e_1)\\
&= (\lambda d.\mathbf{E}[[\text{(X X)}]](\textit{updateenv}\ [[\text{X}]]\\
&\quad\quad d\ e_0))(\mathbf{E}[[\text{X}]]e_1)\\
&= \mathbf{E}[[\text{(X X)}]](\textit{updateenv}\ [[\text{X}]]\\
&\quad\quad (\mathbf{E}[[\text{LAMBDA (X) (X X)}]]e_0)\ e_0)\\
&= \mathbf{E}[[\text{(X X)}]]e_1\\
&= \ldots
\end{aligned}
$$

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

$$\begin{aligned}&\text{E} ::= \ldots \mid \text{LETREC I} = \text{E}_1 \text{ IN E}_2\\ &\quad\mid \text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3\end{aligned}$$

- IFNULL is conditional on lists

$$\begin{aligned}&\mathbf{E}[[\text{IFNULL E}_1 \text{ THEN E}_2 \text{ ELSE E}_3]] =\\ &\quad \lambda e.\ \text{let } x = (\mathbf{E}[[\text{E}_1]]e) \text{ in}\\ &\qquad \text{cases } x \text{ of}\\ &\qquad\quad \text{is}\mathit{List}(t) \to ((\mathit{null}\ t) \to\\ &\qquad\qquad (\mathbf{E}[[\text{E}_2]]e)\ []\ (\mathbf{E}[[\text{E}_3]]e))\\ &\qquad\quad \text{otherwise} \to \text{in}\mathit{Error}()\\ &\qquad \text{end}\end{aligned}$$

Recursive Definitions

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

  $\mathbf{E}[[\text{LETREC I} = \text{E}_1 \text{ IN E}_2]] = \lambda e.\ \mathbf{E}[[\text{E}_2]]e'$
  where $e' = \textit{updateenv}\ [[\text{I}]]\ (\mathbf{E}[[\text{E}_1]]e')e$

  $\mathbf{E}[[\text{LETREC I} = \text{E}_1 \text{ IN E}_2]] = \lambda e.\ \mathbf{E}[[\text{E}_2]]$
  $(\textit{fix}(\lambda e'.\ \textit{updateenv}\ [[\text{I}]]\ (\mathbf{E}[[\text{E}_1]]e')e))$

- 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*

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

- 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

- $\mathbf{E}[[\text{LET I} = \text{E}_1 \text{ IN E}_2]] = \mathbf{E}[[\ [\text{E}_1/\text{I}]\text{E}_2\ ]]$

  LET X = a_0 IN
      LET Y = X CONS NIL IN
          (HEAD Y) CONS X CONS NIL
  $\Rightarrow$ LET Y = a_0 CONS NIL IN
      (HEAD Y) CONS a_0 CONS NIL
  $\Rightarrow$ (HEAD (a_0 CONS NIL)) CONS a_0 CONS NIL
  $\Rightarrow$ a_0 CONS a_0 CONS NIL

- $\mathbf{E}[[\text{LETREC I} = \text{E}_1 \text{ IN E}_2]] = ?$
  - 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.