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 = \((\text{Denotable-value} \rightarrow \text{Denotable-value}) + \text{Denotable-value}^* + \text{Atom} + \text{Error}) \bot\).

• 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

\[ \text{LET } F = a_0 \text{ IN} \]
\[ \text{LET } F = \text{LAMBDA} (Z) F \text{ CONS } Z \text{ IN} \]
\[ \text{LET } Z = a_1 \text{ IN} \]
\[ F(Z \text{ CONS } \text{NIL}) \]

- Context of phrase solely determined by textual position.
- Occurrence of F in function body refers to \( a_0 \)!
- Simplification (see Figure 7.6)
  \[ \Rightarrow (\text{LAMBDA} (Z) a_0 \text{ CONS } Z) (a_1 \text{ CONS } \text{NIL}) \]
  \[ \Rightarrow (a_0 \text{ CONS } (a_1 \text{ CONS } \text{NIL})) \]
Dynamic Scoping

• Context 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:

\[ E[[\text{LAMBDA (I) E}]] = \lambda e. \text{inFunction} \left( \lambda e'. \lambda d. E[[E]](updateenv [[I]] d e') \right) \]

\[ E[[E_1 E_2]] = \lambda e. \text{let } x = (E[[E_1]] e) \text{ in cases } x \text{ of} \]

\[ \text{isFunction}(f) \rightarrow (f e (E[[E_2]] e)) \]

\[ [] \text{ otherwise } \rightarrow \text{inError}() \]

end

Statically scoped languages are difficult to understand.
Example

\[\text{LET } X = a_0 \text{ IN}
\begin{align*}
&\text{(LET } Y = X \text{ CONS NIL IN} \\
&\quad \text{(LET } X = X \text{ CONS } Y \text{ IN } Y)) \\
\Rightarrow &\quad [X \leftarrow a_0]
\end{align*}\]

\[\text{LET } Y = X \text{ CONS NIL IN}
\begin{align*}
&\text{(LET } X = X \text{ CONS } Y \text{ IN } Y)
\Rightarrow &\quad [X \leftarrow a_0] \; [Y \leftarrow X \text{ CONS NIL}] \\
&\quad [X \leftarrow X \text{ CONS } Y]
\end{align*}\]

\[Y\]

\[\Rightarrow &\quad [X \leftarrow a_0] \; [Y \leftarrow X \text{ CONS NIL}] \\
&\quad [X \leftarrow X \text{ CONS } Y]
\]

\[X \text{ CONS NIL}\]

\[\Rightarrow &\quad [X \leftarrow a_0] \; [Y \leftarrow X \text{ CONS NIL}] \\
&\quad [X \leftarrow X \text{ CONS } Y] \\
&\quad (X \text{ CONS } Y) \text{ CONS NIL}
\]

\[\Rightarrow &\quad [X \leftarrow a_0] \; [Y \leftarrow X \text{ CONS NIL}] \\
&\quad [X \leftarrow X \text{ CONS } Y] \\
&\quad (X \text{ CONS } (X \text{ CONS } Y)) \text{ CONS NIL}
\]

\[\Rightarrow \ldots\]
Languages with Contexts II: An Applicative Language

Self-Application

- Untyped applicative language.
- LAMBDA expression can accept itself as argument.
- \(
\text{LET } X = \text{LAMBDA } (X) (X X) \text{ IN } (X X)\)
  \[E[[\text{LET } X = \text{LAMBDA } (X) (X X) \text{ IN } (X X)]]e_0\]
  \[= E[[X (X X)]]e_1 \text{ where} \]
  \[e_1 = (\text{updateenv } [[X]])\]
  \[E[[\text{LAMBDA } (X) (X X)]]e_0) e_0)\]
  \[= E[[\text{LAMBDA } (X) (X X)]]e_0 (E[[X]]e_1)\]
  \[= (\lambda d. E[[((X X)](\text{updateenv } [[X]])\]
  \[d e_0)))(E[[X]]e_1)\]
  \[= E[[((X X)](\text{updateenv } [[X]])\]
  \[E[[\text{LAMBDA } (X) (X X)]]e_0) e_0)\]
  \[= E[[((X X)]e_1\]
  \[= \ldots\]

Circular derivation produced!
Self-Application

• Simplification on semantic expressions is not guaranteed 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 $h(g) = \alpha(g(g))$, $f = h(h)$.

$$f(p) = \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x \times ((pp(p))(x - 1))$$

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

\[ E ::= \ldots | \text{LETREC } I = E_1 \text{ IN } E_2 \\
\quad | \text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3 \]

- IFNULL is conditional on lists

\[ E[[\text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3]] = \lambda e. \text{let } x = (E[[E_1]]e) \text{ in} \]
\[ \begin{align*}
\text{cases } x \text{ of} \\
\text{isList}(t) \rightarrow ((\text{null } t) \rightarrow \\
\quad (E[[E_2]]e) \mathbin{\&\&} (E[[E_3]]e)) \\
\text{otherwise} \rightarrow \text{inError}() \\
\end{align*} \]

end
Recursive Definitions

• Occurrences of $I$ in LETREC refer to the $I$ being declared.

$$E[[\text{LETREC } I = E_1 \text{ IN } E_2]] = \lambda e. \ E[[E_2]]e'$$
where $e' = updateenv \ [[I]] (E[[E_1]]e')e$

$$E[[\text{LETREC } I = E_1 \text{ IN } E_2]] = \lambda e. \ E[[E_2]]$$
$$(\text{fix}(\lambda e'. \ updateenv \ [[I]] (E[[E_1]]e')e))$$

• 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** $G : \text{Environment} \to \text{Environment}$

\[
G^0 = \lambda i. \bot \\
G^1 = \text{updateenv } [[I]] (E[[E_1]](G^0)) \ e \\
= \text{updateenv } [[I]] (E[[E_1]](\lambda i. \bot)) \ e \\
G^2 = \text{updateenv } [[I]] (E[[E_1]](G^1)) \ e \\
= \text{updateenv } [[I]] (E[[E_1]] (\text{updateenv } [[I]] (E[[E_1]](G^1))) \ e)) \ e \\
\ldots \\
G^{i+1} = \text{updateenv } [[I]] (E[[E_1]](G^i)) \ e
\]

- 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!*
Languages with Contexts II: An Applicative Language

Substitution Principles

• \( E[[\text{LET } I = E_1 \text{ IN } E_2]] = E[[E_1/I]E_2] \)

\[
\text{LET } X = a_0 \text{ IN } \\
\text{LET } Y = X \text{ CONS NIL IN } (\text{HEAD } Y) \text{ CONS X CONS NIL } \\
\Rightarrow \text{LET } Y = a_0 \text{ CONS NIL IN } (\text{HEAD } Y) \text{ CONS } a_0 \text{ CONS NIL } \\
\Rightarrow (\text{HEAD } (a_0 \text{ CONS NIL})) \text{ CONS } a_0 \text{ CONS NIL } \\
\Rightarrow a_0 \text{ CONS } a_0 \text{ CONS NIL }
\]

\[
E[[\text{LETREC } I = E_1 \text{ IN } E_2]] = ?
\]

– Substitute \([E_1]\) for \([I]\) in \([E_2]\).
– Substitute \([E_1]\) for \([I]\) in resulting expression.
– Continue until occurrences of \([I]\) eliminated.
– Number of substitutions is unbounded!
– Substitute occurrences only when required for further simplification!

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