Language Design Principles III: Correspondence and Qualification

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

Correspondence

For any parameter binding mechanism, there may exist a corresponding definition mechanism, and vice versa.

- Elements from domain D may be denotable values of formal parameter identifiers.
- Elements from D may be also denotable values of declared identifiers.
- Declared identifiers are used no differently than parameter identifiers.

Compare semantic equations.

Parameters and Definitions

D[[**proc** $\mathsf{I}_1(\mathsf{I}_2)$ = C]] = $\lambda e.\lambda s.$
 ((*updateenv* [[I_1]]
  in*Proc*($\lambda d.$ **C**[[C]]
   (*updateenv* [[I_2]] in*D*(d) e)) e),
 (*return* s))
C[[I(M)]] = $\lambda e.\lambda s.$
 cases (*accessenv* [[I]] e) of
  ...
  [] is*Proc*(q) $\rightarrow$ (q (... **M**[[M]] ...) s)
 end

D[[**define** I=M]] = $\lambda e.\lambda s.$
 ((*updateenv* [[I]] in*D*(... **M**[[M]] ...) e),
 (*return* s))

C[[**begin proc** I(I') = C; I(M) **end**]] =
C[[**begin define** I' = M; C **end**]]

(provided that [[M]] and [[C]] do not contain free occurences of [[I]])

Variable Definitions

D[[**var** I:T]] $= \lambda e.\lambda s.$
let $(d, p) =$ (**T**[[T]]$e\ s$)
in ((*updateenv* [[I]] $d\ e$), p)

Binding of [[I]] is activated type denotation i.e. a reference to newly allocated storage.

D[[**proc** I$_1$(I$_2$) = C]] $= \lambda e.\lambda s.$
((*updateenv* [[I$_1$]]
in*Proc*($\lambda d.$**C**[[C]](*updateenv* [[I$_2$]] $d\ e$)) e),
(*return* s))
C[[I(T)]] $= \lambda e.\lambda s.$
cases (*accessenv* [[I]] e) of
...
[] is*Proc*(q) $\rightarrow$
let $(d, p) =$ (**T**[[T]]$e\ s$)
in (*check* ($q\ d$))(p)
end

By parameter transmission, local variable of type [[T]] is allocated for procedure.

Qualification

Every syntax domain may have a block construct for admitting local declarations.

E ::= ...| **begin** D **within** E **end** | ...
D ::= ...| **begin** D_1 **within** D_2 **end** | ...
T ::= ...| **begin** D **within** T **end** | ...

M: M $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ (*M* $\times$ *Poststore*)
M[[**begin** D **within** M **end**]] $= \lambda e.\lambda s.$
let $(e', p) =$ (**D**[[D]]$e\ s$)
in (*check* (**M**[[M]]e'))(p)

Same form for expression command blocks, expression blocks, type blocks, etc.

Declaration Blocks

Technical problem

- Scope of $[\![D_1]\!]$ should extend only as far as $[\![D_2]\!]$
- Additions by **D** are retained beyond scope of block.

D: Declaration $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$
((Identifier $\times$ *Denotable-value*)* $\times$ *Poststore*)

- Denotation of a declaration is list of binding pairs.
- (**D**$[\![D]\!]e\ s$) denotes the bindings defined by $[\![D]\!]$ and not the environment that results when those bindings are added to e.

Modules

Modula-2.

```
module STACK-OF-NAT =
    begin
        var ST: array [1..k] of nat;
        var TOP: nat
    within
        proc PUSH(I: nat) =
            if TOP=k then skip
            else (TOP := TOP+1; ST[TOP]:=I);
        proc POP = if TOP=0 then skip
            else TOP := TOP-1;
        fcn TOP = if TOP=0 then error
            else ST[TOP];
        proc INITIALIZE = TOP:=0
    end
```

A module is a declaration block abstract!
(Storage is allocated when module is declared.)

Classes

SIMULA, C++.

```
class STACK-OF-(X) =
    record
        begin
            var ST: array [1..k] of nat;
            var TOP: nat
        within
            proc PUSH(I: nat) = ...
            proc POP = ...
            fcn TOP = ...
            proc INITIALIZE = ...
        end
    end
```

var A: STACK-OF(**nat**)

Classes are record type abstracts that enclose declaration blocks.
(Storage is allocated when class variable is defined.)

Orthagonality

- Principles are tools for language design.
- Suggest lines for language extensions.
- Encourage *orthagonal languages*
  - Small number of core concepts.
  - Ways of uniformly combining these concepts.
  - Semantics of combinations are uniform i.e. there are no special restrictions for specific instances of combinations.
- Example: function result types
  - Pascal: only scalar types.
  - ML: any type.

Orthagonality reduces mental overhead for understanding a language, as well for its design and implementation as for its use.