# Language Design Principles I: Abstraction

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

# Language Design Principles

- Abstraction

  Any syntax domain of a language may have definition and invocation mechanisms for abstracts (named entities).

- Parameterization

  A formal parameter to an abstract may be from any syntax domain.

- Correspondence

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

- Qualification

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

*Many important language constrcuts are derived from these principles: subroutines, parameters, block-constructs, . . .*

# Abstraction

Examples:

- Specification that hides computational details.

  Abstract datatypes.

- Function expressions

  $(\lambda x.M)$

- Named abstractions
  - Definition: square = ($\lambda n.$ $n$ times $n$),
  - Invocation: square(two).
- Pascal procedure
  - Abstraction of a command.
  - Invoked by mentioning its name.

# Abstractions

- Definition mechanism.
- Invocation mechanism.
- *Abstract* = *name* plus *body*
  - Body expression from syntax domain $B$.
  - Abstract can be invoked by using its name any place where a $B$ expression is (syntactically) legal.
- Examples:
  - Command abstracts (procedures).
  - Expression abstracts (functions, constants).
  - Declaration abstracts (classes, modules).
  - Type abstracts (named types).

*Any syntax domain of a language may have definition and invocation mechanisms for abstracts.*

## Abstractions

Abstract [[**define** I=V]].

- Name [[I]].
- Body [[V]].
- Denotable value **V**[[V]].
- Question: when is body evaluated?
  - At point of definition?
  - At point of invocation?

# Command Abstracts

Command abstracts = *procedures*.

D ::= $D_1$; $D_2$ | **var** I:T | **proc** I=C
C ::= $C_1$; $C_2$ | L:=E | **begin** D;C **end** | I | ...

Denotation of abstract's body:

- **C**[[C]]: *Environment* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

  Environment and store are provided at the point of invocation (dynamic scoping).

- (**C**[[C]]$e$): *Store* $\rightarrow$ *Poststore*$_\perp$

  Environment provided at the point of definition, store provided at point of invocation (static scoping).

- (**C**[[C]]$e\ s$) $\in$ *Poststore*$_\perp$

  Procedure completely evaluated at point of definition and $[[I]]$ is bound to result store.

*Possible scoping mechanisms.*

# Command Abstracts

## Static scoping.

*Proc* = *Store* $\rightarrow$ *Poststore*$_\perp$
*Denotable-value* = *Natlocn* + *Array* + *Record* +
*Proc*

**D**[[**proc** I=C ]] = $\lambda e.\lambda s.$
((*updateenv* [[I]] in*Proc*(**C**[[C]]$e$) $e$), (*return* $s$))
**C**[[I]] = $\lambda e.\lambda s.$
cases (*accessenv* [[I]] $e$) of
is*Natlocn*($l$) $\rightarrow$ (*signalerr* $s$)
...
[] is*Proc*($q$) $\rightarrow$ ($q$ $s$)
end

***C**[[C]]$e$ is bound to [[I]] in environment, store is supplied at invocation-time.*

# Expression Abstracts

Expression abstracts = *functions*.

D ::= ... | **fcn** I=E
E ::= $E_1$+$E_2$ | ... | I

- Constant definition.

  Environment and store are bound at point of definition.

- Function definition.

  Only environment is bound at point of definition, store is supplied at invocation-time.

- Macro definition.

  Both environment and store are supplied at invocation-time.

*Constants, functions, macros are variations of the same mechanism!*

# Function Procedures

Hybrid of procedure and function.

D ::= ... | **fcnproc** I=C **resultis** E

- Command abstract.
- Invoked by an Expression-identifier.
- Returns an expressible value.
- Expression can alter the value of the store!
- Function procedure may not terminate!

# Function Procedures

**E**: Expression $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$
$(\mathit{Expressible\text{-}value} \times \mathit{Poststore})_\perp$

**D**[[**fcnproc** I=C **resultis** E]] $= \lambda e.\lambda s.$
((*updateenv* [[I]]
in*Fcn-proc*((*check* (**E**[[E]]$e$)) $\circ$
(**C**[[C]]$e$)) $e$),
(*return* $s$))
**E**[[I]] $= \lambda e.\lambda s.$
cases (*accessenv* [[I]] $e$) of
is*Natlocn*($l$) $\rightarrow$ ((*access* $l$ $s$), (*return* $s$))
...
[] is*Fcn-proc*($f$) $\rightarrow$ ($f$ $s$)
end

*All semantic equations for* **E** *must be revised to cope with side-effects and non-termination!*

# Declaration Abstracts

D ::= $D_1$; $D_2$ | **var** I:T | ... | **class** I=D | I

- Invocation of a class activates declarations in its body.
- Multiple invocations of same class in a block lead to a redefinition error.

  Renaming of declarations required.

*Modules are similar to classes (see continuation course).*

# Type Abstracts

D ::= ... | **type** I=T
T ::= **nat** | ... | I

**D**[[**type** I=T]] = $\lambda e.\lambda s.$
((*updateenv* [[I]] in*Type*(**T**[[T]]$e$) $e$),
(*return* $s$))

**T**: Type-structure → *Environment* → *Store* →
(*Denotable-value* × *Poststore*)
**T**[[I]] = $\lambda e.\lambda s.$
cases (*accessenv* [[I]] $e$) of
is*Natlocn*($l$) → (in*Errvalue*(),
(*signalerr* $s$))
...
[] is*Type*($v$) → ($v$ $s$)
end

*When are two variables equivalent in type?*

# Type Equivalence

- Structure equivalence.

  Two variables are type-equivalent if they have identical storage structures.

- Occurence equivalence (name equivalence).

  Two variables are type-equivalent if they are defined with the same occurence of a type expression.

*Structure equivalence more natural in denotational semantics.*

# Example

**type** M = **nat**;
**type** N = **array** [1..3] **of** M;
**var** A: **nat**;
**var** B: M;
**var** C: M;
**var** D: N;
**var** E: **array** [2..4] **of nat**

- A and B are structure-equivalent but not occurence-equivalent.
- B and C are both structure- and occurence-equivalent.
- C and D are neither structure-nor occurence-equivalent.
- Are D and E structure-equivalent?
  - Different range bounds in environment.
  - Similar problems for record types.

# Recursive Bindings

D ::= ... | **rec abs** I=M | ...
**abs** $\in$ {**proc**, **fun**, **class**, ...}

**D**[[**rec abs** I=M]] = $\lambda e.\lambda s.\ (e'$, (*return* $s$))
where $e'$ = (*updateenv* [[I]] in$M$(**M**[[M]]$e'$) $e$)

Means for terminating recursive invocations.

E ::= $E_1$+$E_2$ | ... | **if** $E_1$ **then** $E_2$ **else** $E_3$
T ::= **nat** | ... | **if** E **then** $T_1$ **else** $T_2$

*Which constructions are chosen, depends on language pragmatics.*