Language Design Principles II: Parameterization

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

Parameterization

Abstracts usually carry parameters.

- Parameters = dummy identifiers.
- Replaced by values when abstract invoked.
- *Formal* parameters and *actual* parameters.
- Formal parameter [[I]] used in positions of B-constructs.
- Actual parameter bound to [[I]] must be from syntax domain B.
- Expression parameters, command parameters, type parameters, ...

Formal parameter to an abstract may be from any syntax domain.

Denotation of Parameters

Abstract with body [[V]] and parameters [[I_1]] ... [[I_n]] has denotation of form

$$(\lambda p_1. \ldots . \lambda p_n. \mathbf{V}[[\mathsf{V}]] \ldots)$$

D ::= ... | **proc** $\mathsf{I}_1(\mathsf{I}_2)$ = C
C ::= C_1; C_2 | ... | I(E)

Proc = *Param* → *Store* → *Poststore*$_\perp$

D[[**proc** $\mathsf{I}_1(\mathsf{I}_2)$ = C]] = $\lambda e.\lambda s.$
  ((*updateenv* [[I_1]]
    in*Proc*($\lambda a.$ **C**[[C]](*updateenv* [[I_2]] a e)) e),
  (*return* s))

C[[I(E)]] = $\lambda e.\lambda s.$
  cases (*accessenv* [[I]] e) of
    is*Natlocn*(I) → (*signalerr* s)
    ...
    [] is*Proc*(q) → (q $\underline{(\ldots \mathbf{E}[[\mathsf{E}]] \ldots)}$ s)
  end

Parameter Domains

1. $(\mathbf{E}[[\mathrm{E}]]e\ s) \in$ *Expressible-value*
   - Actual parameter evaluated with environment and store active at point of invocation.
   - Call-by-value.
2. $(\mathbf{E}[[\mathrm{E}]]e)$: *Store* $\rightarrow$ *Expressible-value*
   - Actual parameter uses invocation environment.
   - Uses store active at occurences of formal parameters in procedure body.
   - Call-by-name.
3. $\mathbf{E}[[\mathrm{E}]]$: *Environ.* $\rightarrow$ *Store* $\rightarrow$ *Expr.-value*
   - Uses environment and store active at occurences of formal parameters in procedure body.
   - Call-by-text.
4. *Param* $=$ *Location*
   - Actual parameter must be variable.
   - Denotation is variable's *L-value*.
   - Call-by-reference.

Argument Evaluation

1. Immediate evaluation

Value of actual parameter is calculated before its binding to the formal parameter.

D[[**proc** $\mathsf{I}_1(\mathsf{I}_2) = \mathsf{C}$]] $= \lambda e.\lambda s.$ ((*updateenv* [[I_1]] in*Proc*($\underline{\lambda} a.$ **C**[[C]](*updateenv* [[I_2]] a e)) e), (*return* s))

2. Delayed evaluation

Value of actual parameter need be calculated only upon its use in the procedure body.

- *Call-by-value* = immediate evaluation to an expressible value.
- *Call-by-need* = delayed evaluation to an expressible value.
- Other options use immediate evaluation.

Questions for all parameter domains.

Parameterized Type Expressions

Intersting parameterized abstract.

D ::= ... | **type** $I_1(I_2)$ = T | ...
T ::= ... | I(T) | ...

type STACKOF(T) = **record**
    var ST: **array** [1..k] **of** T;
    var TOP: **nat**
end

var X: STACKOF(**nat**)

Semantics can be constructed analogously to that of procedures.

Polymorphism and Typing

Polymorhpic operations

- Take arguments from more than one semantic domain.
- Produced answer depends upon domains of arguments.
- Example: polymorphic addition on integers and rationals.

  $$(\text{Integer} \times \text{Integer}) \cup (\text{Rational} \times \text{Rational}) \rightarrow \text{Integer} \cup \text{Rational}$$

- Problem: dependence of codomain on domain is not explicitly stated.

Polymorphic operations do not fit cleanly into our semantic notation.

Kinds of Polymorphism

- Ad hoc polymorphism (overloading)
  - Operator "behaves differently" for arguments of different types.
  - Pascal + on integers, reals and for set union.
  - C++ overloading.
- Polymorphic polymorphism
  - Operatore "behaves the same" for all types.
  - ML operations on α-lists.

Formal definition difficult.

Overloaded Operators

Pascal addition expression.

$\mathbf{E}[[\mathrm{E}_1{+}\mathrm{E}_2]] = \lambda e.\lambda s.$
cases $(\mathbf{E}[[\mathrm{E}_1]]e\ s)$ of
is*Nat*$(n_1) \rightarrow$ (cases $(\mathbf{E}[[\mathrm{E}_2]]e\ s)$ of
is*Nat*$(n_2) \rightarrow$ in*Nat*$(n_1$ *plus* $n_2)$
. . . end)
. . .
[] is*Rat*$(r_1) \rightarrow$ (cases $(\mathbf{E}[[\mathrm{E}_2]]e\ s)$ of
. . .
[] is*Rat*$(r_2) \rightarrow$ in*Rat*$(r_1$ *addrat* $r_2)$
. . . end)
. . .
[] is*Set*$(t_1) \rightarrow \ldots$ in*Set*$(t_1$ *union* $t_2) \ldots$
. . .
end

Pre-execution analysis → actual operation can be determined at compile-time.

Parametric Polymorphic Operations

ML **hd** operator.

$$\begin{aligned}\text{Exprval} = (&\text{Nat} + \text{Exprval}^* + (\text{Exprval} \times \text{Exprval}) + \\ &(\text{Exprval} + \text{Exprval}) + (\text{Exprval} \to \text{Exprval}) + \\ &\text{Errvalue})_\perp\end{aligned}$$

$$\begin{aligned}&\mathbf{E}[[\textbf{hd E}]] = \lambda e. \\ &\quad \text{let } x = \mathbf{E}[[\text{E}]]e \text{ in} \\ &\quad\quad \text{cases } x \text{ of} \\ &\quad\quad\quad \text{is}\mathit{Nat}(n) \to \text{in}\mathit{Errvalue}() \\ &\quad\quad\quad [] \text{ is}\mathit{Exprval}^*(l) \to \mathit{hd}\ l \\ &\quad\quad\quad [] \ldots \\ &\quad\quad \text{end}\end{aligned}$$

Polymorphic Parameterized Abstracts

- Previous parameterized abstracts were untyped.
- In typed languages, untyped parameterized abstracts would act polymorphically.
- Formal parameters must be labeled with type expressions.
  - Label acts as precondition or guard.
  - Only actual parameters whose type structures match those of formal parameters are allowed as arguments.

Typed Parameters

D ::= ... | **proc** $I_1(I_2{:}T) = C$
C ::= ... | I(E) | ...

Two ways to handle semantics of typed parameters:

1. Type information can guard entry to abstract at invocation.
2. Abstract's denotation is restricted at definition to a function whose domain is that specified by the type.

1. Version

Type-equivalence checker:

T': Type-structure $\rightarrow$ *Environment* $\rightarrow$
(*Expressible-value* + *Errvalue*)

(**T'**[[T]]$e\ x$)

- Determines whether x has type [[T]].
- If yes, result is *inExpressible-value*(x).
- If no, result is *inErrvalue*().

D[[**proc** I_1(I_2:T) = C]] = $\lambda e.\lambda s.$
((*updateenv* [[I_1]]
in*Proc*($\lambda x.$ cases (**T'**[[T]]$e\ x$) of
is*Expressible-value*(x) $\rightarrow$
C[[C]](*updateenv* [[I]] $x\ e$)
[] is*Errvalue*() $\rightarrow$ *signalerr* end)
e), (*return* s))

2. Version

Family of procedure domains:

Nat-proc = Nat → Store → Poststore$_\perp$
Array-proc = Array → Store → Poststore$_\perp$
Record-proc = Proc → Store → Poststore$_\perp$

Type-tag =
    Nat-tag + Array-tag + Record-tag + Err-tag
    where Nat-tag = ... = Unit
T': Type-structure → Environment → Type-tag

D[[**proc** I$_1$(I$_2$:T) = C]] = $\lambda e.\lambda s.$
    cases (**T'**[[T]]e) of
        is*Nat-tag*() →
            ((*updateenv* [[I$_1$]] in*Nat-proc*($\lambda n.$
                C[[C]](*updateenv* [[I$_2$]]
                    in*Nat*(n) e)) e),
            (*return* s))
        is*Array-tag*() → ...
        is*Record-tag*() → ...
        is*Err-tag*() → (e, (*signalerr* s))
    end