12. Language Design Principles II: Parameterization

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 $[[\mathsf{I}_1]]$ ... $[[\mathsf{I}_n]]$ has denotation of form

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

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

$\mathit{Proc} = \mathit{Param} \to \mathit{Store} \to \mathit{Poststore}_\perp$

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

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

Parameter Domains

1. (**E**[[E]]$e\ s$) $\in$ *Expressible-value*
   - *Actual parameter evaluated with environment and store active at point of invocation.*
   - *Call-by-value.*
2. (**E**[[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. **E**[[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.

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

- 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.

$$
\begin{array}{l}
\mathbf{E}[[\mathsf{E}_1{+}\mathsf{E}_2]] = \lambda e.\lambda s. \\
\quad \text{cases } (\mathbf{E}[[\mathsf{E}_1]]e\ s) \text{ of} \\
\qquad \text{is}\mathit{Nat}(n_1) \rightarrow (\text{cases } (\mathbf{E}[[\mathsf{E}_2]]e\ s) \text{ of} \\
\qquad\quad \text{is}\mathit{Nat}(n_2) \rightarrow \text{in}\mathit{Nat}(n_1\ \mathit{plus}\ n_2) \\
\qquad\quad \ldots \text{end}) \\
\qquad \ldots \\
\qquad [] \text{ is}\mathit{Rat}(r_1) \rightarrow (\text{cases } (\mathbf{E}[[\mathsf{E}_2]]e\ s) \text{ of} \\
\qquad\quad \ldots \\
\qquad\quad [] \text{ is}\mathit{Rat}(r_2) \rightarrow \text{in}\mathit{Rat}(r_1\ \mathit{addrat}\ r_2) \\
\qquad\quad \ldots \text{end}) \\
\qquad \ldots \\
\qquad [] \text{ is}\mathit{Set}(t_1) \rightarrow \ldots \text{in}\mathit{Set}(t_1\ \mathit{union}\ t_2) \ldots \\
\qquad \ldots \\
\quad \text{end}
\end{array}
$$

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

Parametric Polymorphic Operations

ML **hd** operator.

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

E[[**hd** E]] $= \lambda e.$
  let $x =$ **E**[[E]]e in
    cases x of
      is$\mathit{Nat}(n) \to$ in$\mathit{Errvalue}()$
      [] is$\mathit{Exprval}^*(l) \to \mathit{hd}\ l$
      [] ...
    end

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* in*Expressible-value*(x).
- *If no, result is* in*Errvalue*().

D[[**proc** $\mathsf{I}_1(\mathsf{I}_2{:}\mathsf{T}) = \mathsf{C}$]] = $\lambda e.\lambda s.$
((*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* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$
Array-proc $=$ *Array* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$
Record-proc $=$ *Proc* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

Type-tag $=$
    Nat-tag $+$ *Array-tag* $+$ *Record-tag* $+$ *Err-tag*
    where *Nat-tag* $= \ldots =$ *Unit*
T': Type-structure $\rightarrow$ *Environment* $\rightarrow$ *Type-tag*

D[[**proc** I_1(I_2:T) $=$ C]] $= \lambda e.\lambda s.$
    cases (**T'**[[T]]e) of
        is*Nat-tag*() $\rightarrow$
            ((*updateenv* [[I_1]] in*Nat-proc*($\lambda n.$
                C[[C]](*updateenv* [[I_2]]
                    in*Nat*(n) e)) e),
            (*return* s))
        is*Array-tag*() $\rightarrow$ ...
        is*Record-tag*() $\rightarrow$ ...
        is*Err-tag*() $\rightarrow$ (e, (*signalerr* s))
    end