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 [[I_1]] ...[[I_n]] has denotation of form ($\lambda p_{1\ldots}$ λ p_n .V[[V]] ...)

D ::= . . . | proc $I_1(I_2)$ = C
C ::= C 1 ; C 2 | . . . | I(E)

Proc = Param $\rightarrow$ Store $\rightarrow$ Poststore$\bot$

D[[proc $I_1(I_2)$ = C]] = λe λ s.
((updateenv [[I_1]]
inProc(λ a. C[[C]](updateenv [[I_2]] a e)) e),
(return s))

C[[I(E)]] = λe λ s.
cases (accessenv [[I]] e) of
isNatlocn(I) $\rightarrow$ (signalerr s)…
[] isProc(q) $\rightarrow$(q (...E[[E]]...) 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 pa- rameters 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.

$D[[\text{proc } I_1(I_2) = C]]$

$= \lambda e\ \lambda s.\ ((\text{updateenv } [[I_1]]$

$\text{inProc}(\lambda a.C[[C]](\text{updateenv } [[I_2]] a\ e))\ e), (\text{return } s))$

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

$D ::= \ldots \mid \text{type } I_1(I_2) = T \mid \ldots$

$T ::= \ldots \mid I(T) \mid \ldots$

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.
  - (Integer x Integer) U (Rational x Rational) → Integer U 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
  - Operator "behaves the same" for all types.
  - ML operations on α-lists.

Overloaded Operators

E[[E 1 +E 2]] = λe λ s. cases (E[[E1]]e s) of
isNat(n1) → (cases (E[[E 2]]e s) of
isNat(n2) → inNat(n 1 plus n 2) . . . end)
. . .
[] isRat(r1) → (cases (E[[E 2]]e s) of
. . . [] isRat(r2) → inRat(r1 addrat r 2)
. . . end)
. . . [] isSet(t 1) → . . . inSet(t1 union t 2) . . . end

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

Parametric Polymorphic Operations

ML hd operator.

Exprval = (Nat + Exprval* + (Exprval X Exprval) + (Exprval + Exprval) + (Exprval → Exprval) + Errvalue)⊥

E[[hd E]] = λe:

let x = E[[E]]e in cases x of

isNat(n) → inErrvalue()

[] isExprval* (l) → 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.

Type-equivalence checker:

- T': Type-structure → Environment → (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]]

  = λe λs. ((updateenv [[I_1]]

  inProc(λ x: cases (T'[[T]]e x) of

  isExpressible-value(x) →

  C[[C]](updateenv [[I]] x e)

  [] isErrvalue() → signalerr end) e),

  (return s))

Family of procedure domains:

Nat-proc = Nat → Store → Poststore⊥

Array-proc = Array → Store → Poststore⊥

Record-proc = Proc → Store → Poststore⊥

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]] = λe λ s.
  cases (T'[[T]]e) of
  isNat-tag() → ((updateenv [[I_1]] inNat-proc(λn.
  C[[C]](updateenv [[I_2]]
  inNat(n) e)) e), (return s))
  isArray-tag() → . . .
  isRecord-tag() → . . .
  isErr-tag() → (e, (signalerr s))
end

Correspondence

- For any parameter binding mechanism, there may exist a corresponding definition mechnanism, 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.

Qualification

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

E ::= …| begin D within E end |...

D ::= …| begin D1 within D 2 end | . . .

T ::= . . . | begin D within T end | . . .

M: M → Environment → Store → (M x Poststore)

M[[begin D within M end]] = λe λ 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.

Classes

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