The Qualification Principle

Rudolf Mühlbauer

Johannes Kepler Universität, Linz, Austria

2010/01/11

From David A. Schmidt: The Structure of Typed Programming Languages

The Qualification Principle

Any semantically meaningful syntactic class may admit local definitions.

The block form

$$U ::= \cdots \mid \textbf{begin}\ \ D\ \ \textbf{in}\ \ U\ \ \textbf{end}$$

with:

$$\frac{\pi_1 \vdash D : \pi_2 dec \qquad \pi_1 \uplus \pi_2 \vdash U : \theta}{\pi_1 \vdash\ \textbf{begin}\ \ D\ \ \textbf{in}\ \ U\ \ \textbf{end}\ \ : \theta}$$

Typing rules for blocks

$$\frac{\pi \vdash D : \pi_1\, dec \qquad \pi \uplus \pi_1 \vdash C : comm}{\pi \vdash \textbf{ begin } D \textbf{ in } C \textbf{ end } : comm}$$

(Command Block)

$$\frac{\pi \vdash D_1 : \pi_1\, dec \qquad \pi \uplus \pi_1 \vdash D_2 : \pi_2\, dec}{\pi \vdash \textbf{ begin } D_1 \textbf{ in } D_2 \textbf{ end } : \pi_2\, dec}$$

(Declaration Block)

$$\frac{\pi \vdash D : \pi_1\, dec \qquad \pi \uplus \pi_1 \vdash T : \delta\, class}{\pi \vdash \textbf{ begin } D \textbf{ in } T \textbf{ end } : \delta\, class}$$

(Type Structure Block)

(also Actual-list, Expression and Identifier-expression)

Command Blocks

$$\frac{\pi \vdash D : \pi_1\, dec \qquad \pi \uplus \pi_1 \vdash C : comm}{\pi \vdash \textbf{begin } D \textbf{ in } C \textbf{ end } : comm} \quad \text{(Command Block)}$$

```
begin
  var A:newint;

  proc P = begin var C:newint in C:=@A end;

  proc Q = begin var B:newint; var A:newint
           in A:=1; call P end

in A:=0; call P; call Q end
```

Semantics of the Command Block

$$[\![\pi \vdash \textbf{ begin } D \textbf{ in } C \textbf{ end } : comm]\!]\ e\ s = \mathsf{free}(\mathsf{size\text{-}of}\ s)\ s_2$$
$$\text{where } (e_1, s_1) = [\![\pi \vdash D : \pi_1\, dec]\!]\ e\ s$$
$$\text{and } s_2 = [\![\pi \uplus \pi_1 \vdash C : comm]\!]\ (e \uplus e_1)\ s_1$$

$$\mathsf{Store} = \{\langle n_1, n_2, \ldots, n_m\rangle \,|\, n_i \in \mathsf{Int}, \quad 1 \le i \le m, \quad m \ge 0\}$$

$$\mathsf{allocate} : \mathsf{Store} \to \mathsf{Location} \times \mathsf{Store}$$
$$\mathsf{allocate}\,\langle n_1, n_2, \ldots, n_m\rangle = (loc_{m+1}, \langle n_1, n_2, \ldots, n_m, \mathsf{init}\rangle)$$

$$\mathsf{size\text{-}of} : \mathsf{Store} \to \mathsf{Int}$$
$$\mathsf{size\text{-}of}\,\langle n_1, n_2, \ldots, n_m\rangle = m$$

$$\mathsf{free} : \mathsf{Int} \to \mathsf{Store} \to \mathsf{Store}$$
$$\mathsf{free}\ i\,\langle n_1, n_2, \ldots, n_i, n_{i+1}, \ldots, n_m\rangle = \langle n_1, n_2, \ldots, n_i\rangle, \ \text{if } 0 \le i \le m$$

Static Scoping

- compile time typing
- static environment $e \uplus e_1$
- "holes in the scope"

```
begin
  var A:newint;

  proc P = begin var C:newint in C:=@A end;

  proc Q = begin var B:newint; var A:newint
           in A:=1; call P end

in A:=0; call P; call Q end
```

Dynamic Scoping

- hard/impossible to type
- interpreted languages
- environment as a dictionary
- run-time incompatibility errors
- "Strange programs can be written in a dynamically scoped language, and their semantics are dubious."

$$[\![\textbf{define } I = U]\!] = (\{I = f\}, s), \text{ where } f\ e'\ s' = [\![U]\!]\ e'\ s'$$

$$[\![\textbf{invoke } I]\!]\ e\ s = f\ e\ s, \text{ where } (I = f) \in e$$

Extent - Lifetime of a Variable

- controlled by the block
- problem: escape of local variables
  1. return value is location of a local variable
  2. non-local variable assigned to the location of local variable
  3. location of local variable used in abstraction

```
begin proc P = begin var A:newint;
                     proc Q(X:intexp) = A:=X in Q end
 in (call P)(2) end
```

Extent - Lifetime of a Variable (contd.)

- declaration blocks and modules
  - *public* variables must escape scope
  - "A's extent must go beyond module M"
  - eager evaluation ensures proper lifetime

```
module M = begin var A:newint in
                  { proc INIT = A:=0,
                    proc SUCC = A:=@A+1,
                    fun VAL = @A } end
```

Declaration Blocks

$$\frac{\pi \vdash D_1 : \pi_1\, dec \qquad \pi \uplus \pi_1 \vdash D_2 : \pi_2\, dec}{\pi \vdash \textbf{ begin } D_1 \textbf{ in } D_2 \textbf{ end } : \pi_2\, dec}$$

(Declaration Block)

$$[\![\pi \vdash \textbf{ begin } D_1 \textbf{ in } D_2 \textbf{ end } : \pi_2\, dec]\!]\ e\ s =$$
$$[\![\pi \uplus \pi_1 \vdash D_2 : \pi_2\, dec]\!]\ (e \uplus e_1)\ s_1$$
$$\text{where } (e_1, s_1) = [\![\pi \vdash D_1 : \pi_1\, dec]\!]\ e\ s$$

---

with D one of: D_1, D_2, `var`, `proc`, `class`, `module`, `import`

Declaration Blocks (contd.)

```
module RATIONAL-NUMBER =
begin
  class RAT
    = record var NUM:newint, var DEN:newint end
  in
  proc INIT-RAT(N:intexp, D:intexp, M:RAT)
    = M.NUM:=N; M.DEN:=D,
  proc MULT-RAT(M:RAT, N:RAT, P:RAT)
    = P.NUM:=@M.NUM*@N.NUM; P.DEN:=@M.DEN+@N.DEN,
  fun WHOLE-PART(M:RAT) = @M.NUM div @M.DEN,
  ... end
```

- visibility: implementation / definition
- opaque types

Type Structure Blocks

$$[\![\pi \vdash \textbf{ begin } D \textbf{ in } T \textbf{ end } : \pi_2\, dec]\!]\ e\ s =$$
$$[\![\pi \uplus \pi_1 \vdash T : \delta\, class]\!]\ (e \uplus e_1)\ s_1$$
$$\text{where } (e_1, s_1) = [\![\pi \vdash D_1 : \pi_1\, dec]\!]\ e\ s$$

```
class PERSONAL-STACK=
begin var CTR:newint,
      var STACK:array[1..100] of newint
 in record
  proc PUSH(X:intexp) = if @CTR=100
       then skip else CTR:=@CTR+1; STACK[@CTR]:=X fi,
  proc POP = if @CTR=0 then skip else CTR:=@CTR-1 fi,
  fun TOP = if @CTR=0 then failure else @(STACK[@CTR]) fi,
  proc INIT = CTR:=0
end end
```

Type Structure Blocks (contd.)

- every declared variable has own storage
- arbitrary number of *objects* can be created

```
var A:PERSONAL-STACK
call A.INIT; call A.PUSH(0); call A.POP; ... A.TOP ...

var B:PERSONAL-STACK
```

Inheritance

- embedding classes to build new ones
- inheritance
- visibility of variables is an issue

```
class EXTENDED-STACK =
begin var M:PERSONAL-STACK, var N:newint, ...
 in record proc P = ...
           proc PUSH-EXTENDED(X:intexp) = call M.PUSH(X);
... end end
```

```
class EXTENDED-STACK = inherits PERSONAL-STACK with
begin var N:newint, ...
 in record proc P = ...
... end end
```

Inheritance (contd.)

- scoping
- multiple inheritance

```
class T = record proc P = ... end;
class U = inherits T with record proc P = ... end;
var X:U in call X.P
```

```
class T = record proc P = ... end;
class U = record proc P = ... end;
class R = inherits T, inherits U with ...;
var X:R in call X.P
```

Semantics of Inheritance

$$\frac{\pi \vdash T_1 : \pi_1\, class \qquad \pi \uplus \pi_1 \vdash T_2 : \pi_2 class}{\pi \textbf{ inherits } T_1 \textbf{ with } T_2 : (\pi_1 \uplus \pi_2)\, class}$$

$$[\![\pi \vdash \textbf{ inherits } T_1 \textbf{ with } T_2 : (\pi_1 \uplus \pi_2)\, class]\!]\ e\ s = (f, s)$$

$$\text{where } f\ s' = [\![\pi \uplus \pi_1 \vdash T_2 : \pi_2 class]\!]\ (e \uplus e_1)\ s_1$$

$$\text{and } (e_1, s_1) = [\![\pi \vdash T_1 : \pi_1 class]\!]\ e\ s'$$

Object Oriented Languages and Dynamic Scoping

- objects
- inheritance
- dynamic scoping

```
class NAT = record
  var NUM:newint;
  proc SUCC = NUM:=@NUM+1;
  proc PLUSTWO = call SUCC; call SUCC; end
```

```
class INT = inherits NAT with
  record var ISNEG:newbool;
  proc SUCC =
     if !ISNEG then NUM:=@NUM+1
     else NUM:=@NUM-1;
     if @NUM=0 then ISNEG:=false
  fi fi end
```

Object Oriented Languages and Dynamic Scoping (contd.)

```
class NAT = record
  var NUM:newint;
  proc SUCC = self.NUM:=@self.NUM+1;
  proc PLUSTWO = call self.SUCC; call self.SUCC; end
```

```
class INT = inherits NAT with
  record var ISNEG:newbool;
  proc SUCC =
     if !self.ISNEG then self.NUM:=@self.NUM+1
     else self.NUM:=@self.NUM-1;
     if @self.NUM=0 then self.ISNEG:=false
  fi fi end
```

Object Oriented Languages and Dynamic Scoping (contd.)

```
object N:NAT, object I:INT
```

```
rec-var N:record
  var NUM:newint;
  proc SUCC = N.NUM:=@N.NUM+1;
  proc PLUSTWO = call N.SUCC; call N.SUCC; end
```

```
rec-var I:record
  var NUM:newint;
  proc PLUSTWO = call I.SUCC; call I.SUCC end
  var ISNEG:newbool;
  proc SUCC = if !I.ISNEG then I.NUM:=I.NUM+1
    else I.NUM:=@I.NUM-1;
    if @I.NUM = 0 then I.ISNEG:=false
  fi fi end
```

Object Oriented Languages and Dynamic Scoping (contd.)

- introduce *self* moniker
- use dynamic scoping to select method to invoke

- problem: subtyping
- co- and contra variance
- implicit coercion

The Copy Rule for Blocks

- does not work for eagerly evaluated abstractions
- $\Rightarrow$ does not work for local variables

$$\textbf{define } I_1 = U_1, \ldots, \textbf{ define } I_n = U_n \textbf{ in } C \Rightarrow$$
$$[U_1/I_1, \ldots, U_n/I_n]C$$

$$\textbf{begin } \textbf{ define } I_1 = U_1, \ldots, \textbf{ define } I_n = U_n \textbf{ in } V \textbf{ end } \Rightarrow$$
$$[U_1/I_1, \ldots, U_n/I_n]V$$