The Qualification Principle

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

The Qualification Principle

Any semantically meaningful syntactic class may admit local definitions.

- Local definitions.
  - Restrict scope of abstractions by
- *Block structures*:
  - U ::= ... | **begin** D **in** U **end**
  - Declarations D, block *body* U.
  - D visible only to U.
  - $\dfrac{\pi_1 \vdash \mathsf{D}\colon \pi_2 \mathit{dec} \quad \pi_1 \,\overline{\cup}\, \pi_2 \vdash \mathsf{U}\colon \theta}{\pi_1 \vdash \textbf{begin}\ \mathsf{D}\ \textbf{in}\ \mathsf{U}\ \textbf{end}\colon \theta}$
- Correspondence to parameters:
  - For every parameterized abstraction form, a corresponding qualification form (and vice versa).
  - **define** $\mathsf{I}_1(\mathsf{I}_2{:}\theta)$ = U **in** I_1(V)
  - **begin define** I_2 = V **in** U **end**

Command Blocks

- Typing rule:
  - $$\frac{\pi_1 \vdash \mathsf{D}\text{: } \pi_2 \mathit{dec} \quad \pi_1 \,\overline{\cup}\, \pi_2 \vdash \mathsf{C}\text{: } \mathit{comm}}{\pi_1 \vdash \textbf{begin } \mathsf{D} \textbf{ in } \mathsf{C} \textbf{ end}\text{: } \mathit{comm}}$$
- Example:
  - **begin**
    var *A*: **newint**;
    proc *P* = **begin** **var** *C*: **newint**
    in *C*:=@*A* **end**
    proc *Q* = **begin** **var** *B*: **newint**; *A*: **newint**
    in *A*:=1; **call** *P* **end**
    in *A*:=0; **call** *P*; **call** *Q*; **end**
- Local variables:
  - Only alive within block.
  - Stack-based storage management.

Store Algebra

Store $= \{\langle n_1, n_2, \ldots, n_m\rangle \mid$
$\mathsf{n}_i \in$ *Int*, $1 \leq i \leq m$, $m \geq 0\}$

lookup: *Location* $\times$ *Store* $\times$ *Int*
lookup(*loc*$_i$, $\langle n_1, \ldots, n_i, \ldots, n_m\rangle) = n_i$
update: *Location* $\times$ *Int* $\times$ *Store* $\rightarrow$ *Store*
update(*loc*$_i$, n, $\langle n_1, \ldots, n_i, \ldots, n_m\rangle$)
$= \langle n_1, \ldots, n, \ldots, n_m\rangle$
allocate: *Store* $\rightarrow$ *Location* $\times$ *Store*
allocate $\langle n_1, n_2, \ldots, n_m\rangle$
$=$ (*loc*$_{m+1}$, $\langle n_1, n_2, \ldots, n_m,$ *init*$\rangle$
where *init* is some unknown integer
size-of: *Store* $\rightarrow$ *Int*
size-of $\langle n_1, n_2, \ldots, n_m\rangle =$ *m*
free: *Int* $\rightarrow$ *Store* $\rightarrow$ *Store*
free i $\langle n_1, n_2, \ldots, n_i, \ldots, n_m\rangle = \langle n_1, n_2, \ldots, n_i\rangle$
if $0 \leq i \leq m$

Semantics of Command Block

- Semantics:
  - $[[\pi \vdash$ **begin** D **in** C **end**: *comm*$]]e\ s =$
    free (*size-of* s) s_2
    where $(e_1, s_1) = [[\pi \vdash$ D: π_1*dec*$]]e\ s$
    and $s_2 = [[\pi \mathbin{\overline{\cup}} \pi_1 \vdash$ C: *comm*$]](e \mathbin{\overline{\cup}} e_1)\ s_1$
- Stack-based store management:
  - Stack pointer recorded at block entry.
  - D allocates locations for C.
  - Storage reduced to initial size at block exit.

Scope

- How is an invocation associated with the definition it invokes?
  - **begin**
    var A: **newint**;
    proc P = **begin var** C: **newint**
    in C:=@A **end**
    proc Q = **begin var** B: **newint**; A: **newint**
    in A:=1; **call** P **end**
    in A:=0; <u>**call** P</u>; **call** Q; **end**
  - Does C become 0 or 1?
- *Static Scoping*
  - Invocation of I is associated with definition of I whose scope contains *definition*.
  - Assocation between invocations and definitions can be calculated at *compile-time*!
  - Typing rules based on static scoping.

Dynamic Scoping

- *Dynamic Scoping*
  - *Interpreter* for language makes assocatiation between an invocation and a definition.
  - Assocation can change each time invocation is evaluated.
  - Typing rules difficult or impossible.
  - **begin var** A: **newint**; **proc** $P = A{:=}0$
in begin fun $A = 1$
in call P **end end**

Runtime incompatibility errors may occur!

Semantics of Dynamic Scoping

- Phrase's environment not determined until phrase is evaluated!
  - [[**define** I=U]]$e\ s = (\{$I$=f\},\ s)$
    where $f\ e'\ s' =$ [[U]]$e'\ s'$
  - [[**invoke** I]]$e\ s = f\ e\ s$
    where (I$=f$) $\in e$
  - Environment behaves like store!
- Semantics for untyped phrases
  - Types of phrases cannt be predicated.
  - Well-formedness of semantics in question.
- Strange programs can be written:
  - **begin proc** *P* = **call** *P* **in call** *P* *end*
  - [[...]]$e_0\ s_0 = f(e_0 \cup \{P{=}f\})\ s_0$
  - f must receive itself as argument!

Extent

- *Extent*:
  - Lifetime of a variable.
  - Different from scope!
- Command block:
  - Controls extent of local variables.
  - Locations freed upon block exit.
  - Valid strategy because no location bound to local variable may *escape* from block.
- *Escape* of locations:
  - Answer produced by block contains location.
  - Value assigned to global variable contains location.
  - Actual parameter containing location is transmitted to global abstraction that saves location.

Escaping Locations

- Answer produced by block contains location:
  - **begin proc** P =
    begin var A: **newint**;
        proc Q(X: *intexp*) = A:=X **in** Q **end**
in (**call** P)(2) **end**
  - Answer of P is parameterized procedure.
  - P: *intexp* $\rightarrow$ *comm*!
- Location of A escapes P
  - A bound to loc_i.
  - Answer of P is procedure q that updates loc_i.
  - $q\ v\ s = [[mypi \cup \{A\text{: } intloc,\ X\text{: } intexp\} \vdash A{:=}X\text{: } comm]]$
$(e \cup \{A{=}loc_i,\ X{=}v\})\ s$
$= update(loc_i,\ v,\ s)$
- Current typing system prevents this!

Escaping locations must not be freed!

Escaping Locations

- Introduction of *pointer variables*:
  - Variables may contain locations.
  - Let A be global pointer variable.
  - **begin var** B: **newint in** A:=B **end**
  - B's location escapes from block!
  - Separation of stack and *heap* store.
- Actual parameter containing location transmitted to non-local abstraction.
  - Abstraction might save variable in non-local variable.
  - Lazily evaluated parameters that mention variables.
  - P(@A+1)
  - In current language unevaluated phrases cannot be stored.

Extent in Other Block Forms

- Declaration block (module)
  - **module** M = **begin var** A: **newint in**
    { **proc** $INIT$ = A:=0,
    proc $SUCC$ = A:=@A+1
    fun VAL = @A
    } **end**
- Location for A is allocated when M is evaluated.
- A's location must escape declaration block (body of M)!
  - $[[\pi \vdash \textbf{begin } \mathsf{D}_1 \textbf{ in } \mathsf{D}_2 \textbf{ end}\text{: } \pi_2 dec]] e\ s =$
    $[[\pi \cup \pi_1 \vdash \mathsf{D}_2\text{: } \pi_2 dec]](e \cup e_1)\ s_1$
    where $(e_1, s_1) = [[\pi \vdash \mathsf{D}_1\text{: } \pi_1 dec]] e\ s$
- Locations allocated within D_1 and D_2 are retained.

Declaration Blocks

- Hide details about implementation of a structure.
  - **module** *GLOBAL-STACK* =
    begin
    var CTR: **newint**,
    var STACK: **array** [1...100] **of newint**
    in
    proc *PUSH*(*X*: *intexp*) = ...
    proc *POP* = ...
    fun *TOP* = ...
    proc *INIT* = *CTR*:=0
    end
- Stack allocated on module definition.
- Body of declaration block made visible:
  - **import** *GLOBAL-STACK*
    in call *INIT*; **call** *PUSH(0)*; ...
- Stack variable alive but not in scope!

Type Structure Blocks

- Semantics:
  - $[[\pi \vdash$ **begin** D **in** T **end**: δ*class*$]]e\ s =$
    $[[\pi \cup \pi_1 \vdash$ T: δ*class*$]](e \cup e_1)\ s_1$
    where $(e_1, s_1) = [[\pi \vdash$ D_1: π_1*dec*$]]e\ s$
- Bodies are typically *records*
  - **class** *PERSONAL-STACK* **begin**
    var CTR: **newint**,
    var STACK: **array** [1...100] **of newint**
    in record
    ...
    end end
- Stack allocated on declaration of record variable:
  - **var** *A*: *PERSONAL-STACK*
  - Separate stack for each instance of record type.
- Stack use by qualification:
  - **call** *A.INIT*; **call** *A.PUSH*(0); ...

Objects and Classes

- *Object*:
  - Declared variable of a record-structure block.
  - Centerpiece of *object-oriented languages*.
  - Record field = object *methods*.
- *Embedding* of classes:
  - **class** *EXTENDED-STACK* =
    begin
      var *M*: *PERSONAL-STACK*,
      var *N*: **newint**
    in record
      ...
      proc *PUSH-EXTENDED*(*X*: *intexp*) =
        call *M.PUSH*(*X*)
      ...
    end end

Forwarding of method invocations becomes tiresome!

Inheritance of Class Structures

- *Extension* of classes:
  - **class** *EXTENDED-STACK* =
    inherits *PERSONAL-STACK* **with**
    begin
    ...
    in record
    ...
    end end
  - *EXTENDED-STACK* is *subclass* of *PERSONAL-STACK*.
- Semantics of class inheritance:
  - $$\frac{\pi \vdash \mathsf{T}_1\colon \pi_1 \mathit{class} \qquad \pi \mathbin{\dot\cup} \pi_1 \vdash \mathsf{T}_2\colon \pi_2 \mathit{class}}{\pi \vdash \textbf{inherits } \mathsf{T}_1 \textbf{ with } \mathsf{T}_2\colon (\pi_1 \mathbin{\dot\cup} \pi_2)\mathit{class}}$$
  - $[[\pi \vdash \textbf{inherits } \mathsf{T}_1 \textbf{ with } \mathsf{T}_2\colon (\pi_1 \mathbin{\dot\cup} \pi_2)\mathit{class}]]e\ s$
    $= (f\ s)$
    where $f\ s' =$
    $[[\pi \mathbin{\dot\cup} \pi_1 \vdash \mathsf{T}_2\colon \pi_2 \mathit{class}]](e \mathbin{\dot\cup} e_1)\ s_1$
    and $(e_1, s_1) = [[\pi \vdash \mathsf{T}_1\colon \pi_1 \mathit{class}]]e\ s'$

Inheritance Issues

- Are declarations local to superclass also visible to subclass?
  - *public* inheritance in C++.
  - Violates static scoping laws!
- *Overrriding* of class methods:
  - **class** T = **record proc** $P = \ldots$ **end**;
    class U = **inherits** T
    with record proc $P = \ldots$ **end**;
    var X: U **in call** $X.P$
  - P of U is invoked.
- *Multiple inheritance*
  - **class** T = **record proc** $P = \ldots$ **end**;
    class U = **record proc** $P = \ldots$ **end**;
    class R = **inherits** T, **inherits** U
    with $\ldots$ **end**;
    var X: R **in call** $X.P$
  - Which P is selected?

Object-Oriented Languages

- Redefined class methods via 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; . . . ; **fi**
    end
  - In context *INT*, *PLUSTWO* should call *new SUCC*!
  - In context *NAT*, *PLUSTWO* should call *old SUCC*.

Static scoping does not support this!

Dynamic Scoping and Self

- Reference *self* to current object:
  - **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** ...
    else ...; **fi**
    end
- *self* is unresolved identifier:
  - When class is invoked to declare object, *self* references are resolved to object being declared!

The Nature of Self

object *N*: *Nat*, **object** *I*: *INT*

- Self is a "recursive variable"
  - **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
    var ISNEG: newbool
    proc SUCC =
      if ¬I.ISNEG then ...
      else ...; fi
    end
    ```

- *self* is resolved to *N* respectively *I*!

Problems with Method Redefinition

- Typing rules do not hold any more:
  - **class** *K1* = **record**
    proc $P(X\colon intexp) = \ldots;$
    proc $Q = \ldots$ **call** $self.P(3) \ldots$
    end
    class *K2* = **inherits** *K1* **with record**
    proc $P(X\colon boolexp) = \ldots$
    end
    object A: *K2*
  - Invocation **call** $A.Q$ causes run-time error!
  - Redefinition of method with different typing causes problems.
- Typing discipline enforced by *subtyping*:
  - **call** $P(X\colon NAT) = \ldots$ **call** $X.PLUSTWO \ldots$
  - **call** $P(N)$ is allowed.
  - **call** $P(I)$ should be allowed as well.

Idea: allow subclass whereever superclass is allowed.

Problems with Subtyping

- New type attribute *selftype*
  - **class** *NAT2* = **record**
    var *NUM*: **newint**;
    fun *EQ*(*X*: *selftype*) =
    (@*X.NUM*=@*self.NUM*) **end**
  - **class** *INT2* = **inherits** *NAT2* **with record**
    var *ISNEG*: **newbool**
    fun *EQ*(*Y*: *selftype*) =
    (@*Y.ISNEG*=@*self.ISNEG*) & ... **end**
  - **object** *N2*: *NAT2*; **object** *I2*: *INT2*
- Typings
  - *N2.EQ*: *NAT2* → *boolexp*
  - *I2.EQ*: *INT2* → *boolexp*
- Problem:
  - **proc** *Q*(*Z*:*NAT2*) = ... *Z.EQ*(*N2*) ...
  - **call** *Q*(*I2*)

Subclasses are not subtypes!

Dynamically Scoped Objects

- Class is parameterized on value of *self*:
  - $[[$**class** I=T$]]e\ s = (\{$I$=f\},\ s)$
    where $f\ s'\ m = [[$T$]](e \mathbin{\bar{\cup}} \{self{=}m\})\ s'$
- Meaning of *self* supplied at declaration:
  - $[[$**object** I_1: $I_2]]e\ s = (\{I_1{=}b\},\ s')$
    where $(b,\ s') = f\ s\ b$ and $(I_2{=}f) \in e$
  - b is defined in terms of itself!
- Type structures
  - T ::= **newint** | **record** D **end**
    | **inherits** T_1 **with** T_2 | I
  - $[[$**newint**$]]e\ s = allocate\ s$
  - $[[$**record** D **end**$]]e\ s = [[$D$]]e\ s$
  - $[[$**inherits** T_1 **with** $T_2]]e\ s = (e_1 \mathbin{\bar{\cup}} e_2,\ s_2)$
    where $(e_1,\ s_1) = [[T_1]]e\ s$
    and $(e_2,\ s_2) = [[T_2]](e \mathbin{\bar{\cup}} e_1)\ s_1$
  - $[[$I$]]e\ s = f\ s\ m$
    where $($I$=f) \in e$ and $(self{=}m) \in e$

Subtyping

- Typing system that formalizes subtyping:
  - Type attributes $\theta_1 \leq \theta_2$
  - θ_1 is *subtype* of θ_2
  - Phrase of type θ_1 can be used in any context that demands phrase of type θ_2.
  - $\leq$ is *reflexive* and *transitive*.
- Example: mixed mode arithmetic.
  - *int* $\leq$ *real*
  - $\tau_1 \leq \tau_2 \Rightarrow \tau_1 \mathit{exp} \leq \tau_2 \mathit{exp}$.
- Typing rules:
  - $\dfrac{\pi \vdash \mathrm{E}\colon \theta_1}{\pi \vdash \mathrm{E}\colon \theta_2}$ if $\theta_1 \leq \theta_2$
  - $\dfrac{\pi \vdash \mathrm{E}_1\colon \tau\,\mathit{exp} \quad \pi \vdash \mathrm{E}_2\colon \tau\,\mathit{exp}}{\pi \vdash \mathrm{E}_1 + \mathrm{E}_2\colon \tau\,\mathit{exp}}$
  - $\dfrac{\pi \vdash \mathrm{E}\colon \tau}{\pi \vdash \mathrm{E}\colon \tau\,\mathit{exp}}$ $\tau \in \{\mathit{int}, \mathit{real}\}$
  - $\pi \vdash \mathrm{N}\colon \mathit{int}$ $\qquad \pi \vdash \mathrm{N}_1.\mathrm{N}_2\colon \mathit{real}$

Minimal Types

- *Coercion rule*
  - $\pi \vdash$ 2: *real*
  - $\pi \vdash$ 2+4.5: *realexp*
- Unicity of typing is lost:
  - $\pi \vdash$ 2+3: *intexp*
  - $\pi \vdash$ 2+3: *realexp*
- *Minimal* type of every expression:
  - $\pi \vdash$ U: θ_1
  - $\theta_1 \leq \theta_2$, for all $\pi \vdash$ U: θ_2
  - Attach each tree node with minimal type!
- Minimal type may not exist:
  - *nonnegative* $\leq$ *int*
  - *nonpositive* $\leq$ *int*
  - 0: *nonnegative*, 0: *nonpositive*

Denotational Semantics

- Meaning of subtyping relation
  - $[[\theta_1 \leq \theta_2]]$: $[[\theta_1]] \to [[\theta_2]]$
  - *Coercion* map must exist.
  - $[[\theta \leq \theta]]$ is identity function.
  - $[[\theta_1 \leq \theta_3]] = [[\theta_2 \leq \theta_3]] \circ [[\theta_1 \leq \theta_2]]$
  - $[[\pi \vdash \mathsf{E}\text{: } \theta_2]]e\ s = [[\theta_1 \leq \theta_2]]([[\pi \vdash \mathsf{E}\text{: } \theta_1]]e\ s)$
- Semantics must be *coherent*
  - Meaning of well-typed program independent of derivation used to assert well-typing.
- Subtyping on objects:
  - $\{\mathsf{I}_1{:}\theta_1,\ldots,\mathsf{I}_m{:}\theta_m,\ldots,\mathsf{I}_n{:}\theta_n\} \leq$
    $\{\mathsf{I}_1{:}\theta_1',\ldots,\mathsf{I}_m{:}\theta_m'\}$
    if $0 \leq m \leq n$ and $\theta_j \leq \theta_j'$ for all $1 \leq j \leq m$
  - $[[\{\mathsf{I}_1{:}\theta_1,\ldots,\mathsf{I}_m{:}\theta_m,\ldots,\mathsf{I}_n{:}\theta_n\} \leq$
    $\{\mathsf{I}_1{:}\theta_1',\ldots,\mathsf{I}_m{:}\theta_m'\}]]$
    $\{\mathsf{I}_1{=}v_1,\ldots,\mathsf{I}_m{=}\mathsf{v}_m,\ldots,\mathsf{I}_n{=}\mathsf{v}_n\}$
    $= \{\mathsf{I}_1{=}[[\theta_1{\leq}\theta_1']]v_1,\ldots,[[\theta_m{\leq}\theta_m']]v_m\}$

Extension of Subtyping

- Other parts of programming language:
  - Subtyping on modules.
  - π_1*dec* $\leq$ π_2*dec* if $\pi_1 \leq \pi_2$
  - **fun** A=2, **proc** A=**skip**
  - {A:*intexp*}*dec* $\leq$ {}*dec*
  - {A:*comm*}*dec* $\leq$ {}*dec*
  - {A:*intexp*}*dec* or {A:*comm*}*dec*?
  - Two different incorrect typings!
- Function types:
  - $\theta_{11} \to \theta_{12} \leq \theta_{21} \to \theta_{22}$
    if $\theta_{21} \leq \theta_{11}$ and $\theta_{12} \leq \theta_{22}$
  - *Contravariance* of argument and result types!
- Example:
  - TRUNC: *realexp* $\to$ *intexp*
  - Result can be extended to *realexp*
  - Argument can be extended to *realexp*
  - TRUNC: *intexp* $\to$ *realexp*!

Subtyping of Function Types

Solution to method redefinition problem

- Object types:
  - N2: *rec* n.[n/*selftype*]
    {NUM: *intloc*, EQ: *selftype* $\rightarrow$ *boolexp*}
  - I2: *rec* i.[i/*selftype*]
    {NUM: *intloc*, ISNEG: *boolloc*,
    EQ: *selftype* $\rightarrow$ *boolexp*}
- Resolution of *selftype*
  - n = {NUM: *intloc*, EQ: $n \rightarrow$ *boolexp*}
  - i = {NUM: *intloc*, ..., EQ: $i \rightarrow$ *boolexp*}
  - $i \leq n$ does *not* imply $i\rightarrow$*boolexp*$\leq n\rightarrow$*boolexp*!
  - **call** *Q*(*I2*) is *not* well-typed!
- Methods that take objects as parameters can destroy subtype relation!
  - C++: methods in subclasses must have *same* argument types.

OO typing systems still topic of research!

The Copy Rule for Blocks

- Copy rule for lazy evaluated abstractions also applies when abstractions are locally defined within a block:
  - **begin define** I_1=U_1, ..., **define** I_n=U_n
    in V **end**
    $\Rightarrow [\mathsf{U}_1/\mathsf{I}_1, \ldots, \mathsf{U}_n/\mathsf{I}_n]\mathsf{V}$
- Example:
  - **begin const** A=2, **const** B=3
    in begin fun $F(X{:}\mathit{intexp}) = X{+}A{+}B$
    in begin const A=4 **in** $F(A){+}B$ **end**
    end
    end
  - $\Rightarrow \ldots$
  - $\Rightarrow(\lambda X.\ X{+}2{+}3)(4){+}3$

Does not apply to eagerly evaluated abstractions!

Record Introduction Principle

- The qualification principle is a *record introduction* principle.
- Locally defined abstractions:
  - Type attributes are *record types*.
  - Semantic values are *records of bindings*.
- *Scope* of bindings is crucial concept.
- Block construct:
  - Block **begin** D **in** U **end**
  - Expression *with* D *do* U
- Copy rule is indexing rule:
  - *with* $\{\mathrm{I}_1{=}\mathrm{V}_1, \ldots, \mathrm{I}_n{=}\mathrm{V}_n\}$ *do* U
    $\Rightarrow [\mathrm{V}_1/\mathrm{I}_1, \ldots, \mathrm{V}_n/\mathrm{I}_n]\mathrm{U}$
  - *R.I* = *with R do I*

Programming language consists of core language uniformly extended by records and parameters!

Summary (Syntax)

- Abstract syntax:
  - U ::= ... | **begin** D **in** U **end**
  - D ::= ... | **begin** D_1 **in** D_2 **end**
  - T ::= ... | **begin** D **in** T **end**
  - C ::= ... | **begin** D **in** C **end**
  - D ::= ... | **begin** D **in** E **end**
- Typing rules:
  - $$\frac{\pi \vdash \mathrm{D}\text{: } \pi_1 \textit{dec} \quad \pi \mathbin{\overline{\cup}} \pi_1 \vdash \mathrm{U}\text{: } \theta}{\pi \vdash \textbf{begin}\ \mathrm{D}\ \textbf{in}\ \mathrm{U}\ \textbf{end}\text{: } \theta}$$
  - $$\frac{\pi \vdash \mathrm{D}_1\text{: } \pi_1 \textit{dec} \quad \pi \mathbin{\overline{\cup}} \pi_1 \vdash \mathrm{D}_2\text{: } \pi_2 \textit{dec}}{\pi \vdash \textbf{begin}\ \mathrm{D}\ \textbf{in}\ \mathrm{D}_2\ \textbf{end}\text{: } \pi_2 \textit{dec}}$$
  - $$\frac{\pi \vdash \mathrm{D}\text{: } \pi_1 \textit{dec} \quad \pi \mathbin{\overline{\cup}} \pi_1 \vdash \mathrm{T}\text{: } \delta \textit{class}}{\pi \vdash \textbf{begin}\ \mathrm{D}\ \textbf{in}\ \mathrm{T}\ \textbf{end}\text{: } \delta \textit{class}}$$
  - $$\frac{\pi \vdash \mathrm{D}\text{: } \pi_1 \textit{dec} \quad \pi \mathbin{\overline{\cup}} \pi_1 \vdash \mathrm{C}\text{: } \textit{comm}}{\pi \vdash \textbf{begin}\ \mathrm{D}\ \textbf{in}\ \mathrm{C}\ \textbf{end}\text{: } \textit{comm}}$$
  - $$\frac{\pi \vdash \mathrm{D}\text{: } \pi_1 \textit{dec} \quad \pi \mathbin{\overline{\cup}} \pi_1 \vdash \mathrm{E}\text{: } \tau \textit{exp}}{\pi \vdash \textbf{begin}\ \mathrm{D}\ \textbf{in}\ \mathrm{E}\ \textbf{end}\text{: } \tau \textit{exp}}$$

Summary (Semantics)

- Command block:
  - $[[\pi \vdash$ **begin** D **in** C **end**: *comm*$]]e\ s =$
    free (*size-of* s) s_2
    where $(e_1, s_1) = [[\pi \vdash$ D: π_1*dec*$]]e\ s$
    and $s_2 = [[\pi \uplus \pi_1 \vdash$ C: *comm*$]](e \uplus e_1)\ s_1$
- Blocks without storage management:
  - $[[\pi \vdash$**begin** D **in** U **end**$]]e\ s =$
    $[[\pi \uplus \pi_1 \vdash$ U: $\theta]]e_1\ s_1$
    where $(e_1, s_1) = [[\pi \vdash$ D: π_1*dec*$]]e\ s$
- Copy rule
  - **begin define** I_1=U_1, ..., **define** I_n=U_n
    in U **end**
    $\Rightarrow [\mathsf{U}_1/\mathsf{I}_1, \ldots, \mathsf{U}_n/\mathsf{I}_n]\mathsf{U}$