The Abstraction Principle I

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 Abstraction Principle

The phrases of any semantically meaningful syntactic class may be named.

- Any construct may be named.
  - Location: *alias*.
  - Numeral: *const*.
  - Expression: *function*.
  - Command: *procedure*.
  - Declaration *module*.
  - Type structure: *class (type)*.
- Names important for handling complexity (by decomposing a task into named subtasks).

Examine the role of names in a programming language.

Abstractions

define I = V

- Abstractions introduced by *definitions*.
  - I ... *name*.
  - V ... *body*.
  - Association of I to V ... *binding*.
- Abstractions activated by *invocations*.
  - I
  - **invoke** I
  - **call** I
- Example: command abstractions
  - **proc** I=C.
  - **call** I

Systematic introduction of names into our core language.

Expression Abstractions

Expression abstractions = *functions*.

P ∈ Program	L ∈ Location
D ∈ Declaration	N ∈ Numeral
C ∈ Command	I ∈ Identifier
E ∈ Expression	

P ::= D **in** C
D ::= **fun** I=E | D_1,D_2 | D_1;D_2
C ::= L:=E | C_1;C_2 | **if** E **then** C_1 **else** C_2 **fi**
| **while** E **do** C **od** | **skip**
E ::= N | @L | E_1+E_2 | E_1=E_2 | ¬E | I
L ::= loc_i, $i > 0$
N ::= n, $n \in Int$

- New domains *Identifier* and *Declaration*.
- Definition: **fun** I=E.
- Invocation: I.

Type Attributes of Functions

- Type attribute of a function established by its definition.
- Type attribute forwarded to users of function
- Introduce *type assignment* $\pi = S$*dec*.
  - Set S of (identifier:type attribute) pairs.
  - For each I, at most one (I:θ) $\in S$.
- Example:
  - **fun** A = 1+@loc_1.
  - **fun** B = @loc_1=0.
  - π_1 = {A: *intexp*, B: *boolexp*}*dec*.
- Each tree node annotated with type and type assignment.
  - $\pi_1 \vdash loc_1$:=A+2: *comm*.
  - See Schmidt, Figure 2.2.

Typing Rules

- Program:
  - $$\frac{\emptyset \vdash \text{D: } \pi\textit{dec} \quad \pi \vdash \text{C: } \textit{comm}}{\vdash \text{D } \textbf{in} \text{ C: } \textit{comm}}$$
  - To annotate D **in** C, annotate D-tree in empty assignment $\emptyset$ and obtain its type π*dec*. Then annotate C-tree in π*dec*; if it has type *comm*, annotate D **in** C with type *comm*.
- Declaration:
  - $$\frac{\pi \vdash \text{E: } \tau\textit{exp}}{\pi \vdash \textbf{fun } \text{I=E: } \{\text{I:}\tau\textit{exp}\}\textit{dec}}$$
  - $$\frac{\pi \vdash \text{D}_1\text{: } \pi_1\textit{dec} \quad \pi \vdash \text{D}_2\text{: } \pi_2\textit{dec}}{\pi \vdash \text{D}_1\text{,D}_2\text{: } (\pi_1 \dot\cup \ \pi_2)\textit{dec}}$$
  - $$\frac{\pi \vdash \text{D}_1\text{: } \pi_1\textit{dec} \quad \pi\dot\cup \ \pi \vdash \text{D}_2\text{: } \pi_2\textit{dec}}{\pi \vdash \text{D}_1\text{,D}_2\text{: } (\pi_1 \dot\cup \ \pi_2)\textit{dec}}$$
  - $\pi_1 \dot\cup \ \pi_2 := \pi_1 \cup \pi_2$, if $\{\text{I} \mid (\text{I:}\theta) \in \pi_1\} \cap \{\text{I} \mid (\text{I:}\theta) \in \pi_2\} = \emptyset$ (else undefined)

Construction and combination of type assignments.

Typing Rules

- Command
  - $$\frac{\pi \vdash \mathsf{L}\colon \mathit{intloc} \quad \pi \vdash \mathsf{E}\colon \mathit{intexp}}{\pi \vdash \mathsf{L}{:=}\mathsf{E}\colon \mathit{comm}}$$
  - $$\frac{\pi \vdash \mathsf{C}_1\colon \mathit{comm} \quad \pi \vdash \mathsf{C}_2\colon \mathit{comm}}{\pi \vdash \mathsf{C}_1;\mathsf{C}_2\colon \mathit{comm}}$$
  - $$\frac{\pi \vdash \mathsf{E}\colon \mathit{boolexp} \quad \begin{array}{l}\pi \vdash \mathsf{C}_1\colon \mathit{comm}\\ \pi \vdash \mathsf{C}_2\colon \mathit{comm}\end{array}}{\pi \vdash \mathbf{if}\ \mathsf{E}\ \mathbf{then}\ \mathsf{C}_1\ \mathbf{else}\ \mathsf{C}_2\ \mathbf{fi}\colon \mathit{comm}}$$
  - $$\frac{\pi \vdash \mathsf{E}\colon \mathit{boolexp} \quad \pi \vdash \mathsf{C}\colon \mathit{comm}}{\pi \vdash \mathbf{while}\ \mathsf{E}\ \mathbf{do}\ \mathsf{C}\ \mathbf{od}\colon \mathit{comm}}$$
  - $\pi \vdash \mathbf{skip}\colon \mathit{comm}$

Transportation of type assignments.

Typing Rules

- Expression
  - $\dfrac{\pi \vdash \text{N: } \mathit{int}}{\pi \vdash \text{N: } \mathit{intexp}}$
  - $\dfrac{\pi \vdash \text{L: } \mathit{intloc}}{\pi \vdash \text{@L: } \mathit{intexp}}$
  - $\dfrac{\pi \vdash \text{E}_1\text{: } \mathit{intexp} \quad \pi \vdash \text{E}_2\text{: } \mathit{intexp}}{\pi \vdash \text{E}_1\text{+E}_2\text{: } \mathit{intexp}}$
  - $\dfrac{\pi \vdash \text{E}_1\text{: } \mathit{intexp} \quad \pi \vdash \text{E}_2\text{: } \mathit{intexp}}{\pi \vdash \text{E}_1\text{=E}_2\text{: } \mathit{boolexp}}$
  - $\dfrac{\pi \vdash \text{E: } \mathit{boolexp}}{\pi \vdash \neg\text{E: } \mathit{boolexp}}$
  - $\pi \vdash$ I: τ*exp*, if (I:τ*exp*) $\in \pi$
- Location:
  - $\pi \vdash$ loc_i: *intloc*
- Numeral
  - $\pi \vdash$ n: *int*

Function invocation uses type assignment.

Linear Type Annotation

See Schmidt, FIgure 2.2.

- Linear form of well-typed trees.
  - Type assignments as subscribts.
  - Type attributes as superscripts.
- Example
  - **fun** A=1, **fun** B=(@loc_1=0)
    in while B **do** loc_1 := (A+2) **od**
  - ((**fun** A=$((1)_{\emptyset}^{int})_{\emptyset}^{intexp})_{\emptyset}^{\{A:intexp\}dec}$,
    (**fun** B =
    ($((@(loc_1)_{\emptyset}^{intloc})_{\emptyset}^{intexp}$ =
    $((0)_{\emptyset}^{int})_{\emptyset}^{intexp})_{\emptyset}^{boolexp})_{\emptyset}^{\{B:boolexp\}dec})_{\emptyset}^{\pi_1 dec}$
    in (**while** $(B)_{\pi_1}^{boolexp}$ **do**
    $((loc_1)_{\pi_1}^{intloc}$:=
    $((A)_{\pi_1}^{intexp}+(2)_{\pi_1}^{int})_{\pi_1}^{intexp})_{\pi_1}^{comm}\ )_{\pi_1}^{comm}\ )^{comm}$
  - π= {A: *intexp*, B: *boolexp*}

Interpretation

- $\pi \vdash$ U:θ
  - Within context π, U has type θ.
  - $\pi_1 \vdash$ *loc*$_1$:=A+2: *comm*.
- Type annotations can be constructed by left-to-right post-order tree traversal.
  - While descending tree to leaves, attach type assignments to nodes.
  - While ascending tree to root, attach type attributes to nodes.
- Attribute grammars:
  - Type assignments are *inherited attributes.*
  - Type attributes are *synthesized attributes.*

Typing rules define typing algorithm!

The Semantics of Abstractions

- *Environments* hold function definitions.
  - Set of (identifier, meaning) pairs.
  - Must be propagated to all tree nodes.
- Program: [[P: *comm*]] $\in$ *Store* $\rightarrow$ *Store*$_\perp$
  - [[$\vdash$ D **in** C: *comm*]]s = [[$\pi \vdash$ C: *comm*]]$e_0\ s$
    where e_0 = [[$\emptyset \vdash$D: π*dec*]]$\emptyset\ s$
- Declaration: [[D: S*dec*]]: *Environment* $\rightarrow$ *Store* $\rightarrow$ *Environment*
  - [[$\pi \vdash$ **fun** I=E: $\{$I:τ*exp*$\}$]]$e\ s$ = $\{$I=$f\}$
    where $f(s')$ = [[$\pi \vdash$ E: τ*exp*]]$e\ s'$
  - [[$\pi \vdash$ D$_1$;D$_2$: $(\pi_1 \dot\cup\ \pi_2)$*dec*]]$e\ s = e_1 \cup e_2$
    where e_1 = [[$\pi \vdash$ D$_1$: π_1*dec*]]$e\ s$
    and e_2 = [[$\pi \vdash$D$_2$: π_2*dec*]]$e\ s$
  - [[$\pi \vdash$D$_1$;D$_2$: $(\pi_1 \dot\cup\ \pi_2)$*dec*]]$e\ s = e_1 \cup e_2$
    where e_1 = [[$\pi \vdash$ D$_1$: π_1*dec*]]$e\ s$
    and e_2 = [[$\pi \dot\cup \pi_1 \vdash$ D$_2$: π_2*dec*]]$(e \cup e_1)\ s$

The Semantics of Abstractions

- Command: $[[\mathsf{C}\text{: } \textit{comm}]] \in \textit{Environment} \rightarrow \textit{Store} \rightarrow \textit{Store}_\perp$
  - $[[\pi \vdash \mathsf{L{:=}E}\text{: } \textit{comm}]]e\ s =$
    $\textit{update}([[\pi \vdash \mathsf{L}\text{: } \textit{intloc}]]e,$
    $[[\pi \vdash \mathsf{C}_1\text{: } \textit{comm}]]e\ s,\ s)$
  - $[[\pi \vdash \mathsf{C}_1;\mathsf{C}_2\text{: } \textit{comm}]]e\ s = [[\pi \vdash \mathsf{C}_2\text{: } \textit{comm}]]e$
    $([[\pi \vdash \mathsf{C}_1\text{: } \textit{comm}]]e\ s)$
  - $[[\pi \vdash \textbf{if } \mathsf{E} \textbf{ then } \mathsf{C}_1 \textbf{ else } \mathsf{C}_2 \textbf{ fi}\text{: } \textit{comm}]]e\ s =$
    $\textit{if}([[\pi \vdash \mathsf{E}\text{: } \textit{boolexp}]]e\ s,$
    $[[\pi \vdash \mathsf{C}_1\text{: } \textit{comm}]]e\ s,$
    $[[\pi \vdash \mathsf{C}_2\text{: } \textit{comm}]]e\ s)$
  - $[[\pi \vdash \textbf{while } \mathsf{E} \textbf{ do } \mathsf{C} \textbf{ od}\text{: } \textit{comm}]]e\ s = w(s)$
    where $w(s) =$
    $\textit{if}([[\pi \vdash \mathsf{E}\text{: } \textit{boolexp}]]e\ s,$
    $w([[\pi \vdash \mathsf{C}\text{: } \textit{comm}]]e\ s),\ s)$
  - $[[\pi \vdash \textbf{skip}\text{: } \textit{comm}]]e\ s = s$

The Semantics of Abstractions

- Expression: $[\![\text{E: } \tau\textit{exp}]\!] \in \textit{Environment} \rightarrow \textit{Store} \rightarrow [\![\tau]\!]$
  - $[\![\pi \vdash \text{N: } \textit{intexp}]\!] e\ s = [\![\pi \vdash \text{N: } \textit{int}]\!] e$
  - $[\![\pi \vdash \text{@L: } \textit{intexp}]\!] e\ s =$
    $\textit{lookup}([\![\pi \vdash \text{L: } \textit{intloc}]\!] e,\ s)$
  - $[\![\pi \vdash \text{E}_1\text{+E}_2\text{: } \textit{intexp}]\!] e\ s = \textit{plus}($
    $[\![\pi \vdash \text{E}_1\text{: } \textit{intexp}]\!] e\ s,\ [\![\pi \vdash \text{E}_2\text{: } \textit{intexp}]\!] e\ s)$
  - $[\![\pi \vdash \text{E}_1\text{=E}_2\text{: } \textit{boolexp}]\!] e\ s = \textit{equal}($
    $[\![\pi \vdash \text{E}_1\text{: } \textit{intexp}]\!] e\ s,\ [\![\pi \vdash \text{E}_2\text{: } \textit{intexp}]\!] e\ s)$
  - $[\![\pi \vdash \neg\text{E: } \textit{boolexp}]\!] e\ s =$
    $\textit{not}([\![\pi \vdash \text{E: } \textit{boolexp}]\!] e\ s)$
  - $[\![\pi \vdash \text{I: } \tau\textit{exp}]\!] e\ s$, where $(\text{I}=v) \in e$
- Location: $[\![\text{L: } \textit{intloc}]\!] \in \textit{Environment} \rightarrow \textit{Location}$
  - $[\![\pi \vdash \textit{loc}_i\text{: } \textit{intloc}]\!] e = \textit{loc}_i$
- Numeral: $[\![\text{N: } \textit{int}]\!] \in \textit{Environment} \rightarrow \textit{Int}$
  - $[\![\pi \vdash n\text{: } \textit{int}]\!] e = n$

Soundness of the Typing Rules

Are typing rules and semantics well formed?

- Environment e must be *consistent* with type assignment π
  - Calculation of $[[\pi \vdash \mathrm{U}{:}\theta]]e$.
- Syntax of type attributes:
  - θ ::= τ *exp* | *intloc* | *comm* | π *dec*.
  - τ ::= *int* | *bool*.
  - π ::= $\{\mathrm{i}{:}\theta_i\}_{i \in I}$, where I $\subseteq$ Identifier.
- Meaning of type attributes:
  - $[[\tau\,\mathit{exp}]] = \mathit{Store} \to [[\tau]]$.
  - $[[\mathit{intloc}]] = \mathit{Location}$.
  - $[[\mathit{comm}]] = \mathit{Store} \to \mathit{Store}_{\perp}$.
  - $[[\pi\,\mathit{dec}]] = \mathit{Store} \to [[\pi]]$.
  - $[[\mathit{int}]] = \mathit{Int}$.
  - $[[\mathit{bool}]] = \mathit{Bool}$.
  - $[[\{i{:}\theta_i\}_{i \in I}]] = \{i{:}[[\theta_i]]\}_{i \in I}$
    (set of environments $\{\mathrm{I}_1{=}v_1, \ldots, \mathrm{I}_n{=}v_n\}$).

Soundness Theorem

- *Consistency:*
  - Environment e is *consistent* with type assignment π if (I:θ) $\in \pi$ iff (I=v) $\in e$ and $v \in [[\theta]]$.
- *Soundness Theorem:*
  - If $\pi \vdash$ U:θ and e consistent with π, then $[[\pi \vdash$ U:$\theta]]e \in [[\theta]]$.
- Proof: by structural induction on set of typing rules.
  - For $\pi \vdash$L:=E: *comm*, typing rule demands $\pi \vdash$ L: *intloc* and $\pi \vdash$E: *intexp*. By inductive hypothesis, $[[\pi \vdash$ L: *intloc*$]]e \in$ *Location* and $[[\pi \vdash$ E: *intexp*$]]e \in$ *Store* $\rightarrow$ *Int* for e consistent with π. Hence, $[[\pi \vdash$ L:=E: *comm*$]]e\ s =$ *update*($[[\pi \vdash$ L:*intloc*$]]e$, $[[\pi \vdash$ E: *intexp*$]]e\ s$, s) $\in$ *Store*, implying $[[\pi \vdash$ L:=E: *comm*$]]e \in$ *Store* $\rightarrow$ *Store*$_\perp$.

The typing rules of the extended core language are sound.

Lazy Evaluation and Copy Rule

- Two styles of function evaluations.
  1. *Lazy* (call by name).
  2. *Eager* (call by value).
- Lazy evaluation for abstractions is norm.
  - Body not computed until invocation.
- Example:
  - **fun** F = @loc_1+1
    in loc_1:=0; loc_1:=F; loc_2:=F+2.
  - @loc+1 is computed each time F is invoked.
- Lazy evaluation supports *copy rule*.
  - ...**fun** I=E ...**in** ...I ...
    = ...**in** ...E ...
  - Body of I is copied into places of invocation.
  - loc_1:=0; loc_1:=@loc_1+1; loc_2:=(@loc_1+1)+2.

Copy rule suits our intuition of abstractions as abbreviations!

Declaration Unions

Assume programs uses just D_1,D_2 for declarations.

- Copy rule:
  - **define** I_1=V_1, ..., **define** I_n=V_n **in** U
    $\Rightarrow$ [V_1/I_1, ..., V_n/I_n]U.
  - Bodies V_j are substituted for invocations I_j in U.
  - *Simultaneous* substitutions V_j/I_j.
- Example:
  - **fun** A=1, **fun** B=@loc_1=0
    in while B **do** loc_1:=A+2 **od**
    $\Rightarrow$ [1/A, @loc_1=0/B]
    while B **do** loc_1:=A+2 **od**
    = **while** @loc_1=0 **do** loc_1:=1+2 **od**

Easy to handle.

Declaration Sequences

Consider D_1;D_2 for declarations.

- Second copy rule:
  - (**define** I_1=V_1, ..., **define** I_n=V_n); D
    ⇒ **define** I_1=V_1, ..., **define** I_n=V_n,
    [V_1, ..., V_n/I_n]D.
  - Translation of D_1;D_2 into D_1,D_n by propagation of bindings of D_1 into D_2.
- Example:
  - **fun** A=@loc_2; **fun** B=A+1
    in loc_2:=A+B
    ⇒ **fun** A=@loc_2, [@loc_2/A](**fun** B=A+1)
    in loc_2:=A+B
    = **fun** A=@loc_2, **fun** B=[@loc_2/A](A+1)
    in loc_2:=A+B
    = **fun** A=@loc_2, **fun** B=@loc_2+1
    in loc_2:=A+B
    ⇒ [@loc_2/A, @loc_2+1/B]loc_2:=A+B
    = loc_2:=(@loc_2)+(@loc_2+1)

Substitutions

- Substitution preserves typing of a phrase:
  - If $\pi_1 \dot{\cup} \ \pi_2 \vdash \mathsf{U}{:}\theta$, then $\pi_2 \vdash [\mathsf{V}_j/\mathsf{I}_j]\mathsf{U}{:}\theta$
    where $\pi_1 = \{j{:}\theta_j\}_{j \in I}$ and $\pi_2 \vdash \mathsf{V}_j{:}\theta_j$,
    for all $j \in \mathsf{I}$.
- Proof: induction on typing rules.
- Collorary: copy rule preserves typing
  - Program needs to be type checked only once.
  - Type assignments "predict" substitutions.

Copy rule is device for understanding programs with abstractions by corresponding programs in the core language!

Eager Evaluation

- Copy rule fails on eager evaluations.
  - Body is evaluated when definition is evaluated.
- Example:
  - **fun** $F = @loc_1{+}1$
    in $loc_1{:=}0$; $loc_1{:=}F$; $loc_2{:=}F{+}2$.
  - $@loc{+}1$ is computed when F is defined.
  - Function definition is "const" definition.
- Semantics definition necessary for determining meaning.

"Const" definitions are eagerly evaluated function definitions!

Semantics of Lazy Evaluation

- Store is given when function is invoked.
  - [[$\pi \vdash$ **define** I=U: {I:θ}*dec*]]$e\ s = \{$I$=f\}$,
    where $f s' =$ [[$\pi \vdash$ U:θ]]$e\ s'$.
  - [[$\pi \vdash$ **invoke** I:θ]]$e\ s = f s$,
    where (I$=f$) $\in e$.
- Soundness of copy rule:
  - [[$\vdash$ **define** I_j=V_j **in** C: *comm*]]
    = [[$\emptyset \vdash$ [V_j/I_j]C: *comm*]]$\emptyset$.
- Relation to core language:
  - Core semantics [[.]]$_c$, extended semantics [[.]]$_a$.
  - If C contains no identifiers, then [[C: *comm*]]$_c$ = [[$\pi \vdash$ C: *comm*]]$_a$ e, for e consistent with π.
- Copy rule semantics:
  - [[$\vdash$ **define** I_1=$\mathsf{V}_1, \ldots,$ I_n=V_n **in** C: *comm*]]$_a$ = [[[V_j/I_j]C: *comm*]]$_c$.

Copy rule simpler than basic semantics!

Semantics of Eager Evaluation

- Store is given when function is defined.
  - $[[\pi \vdash$ **define** I=U: $\{$I:$\theta\}$*dec*$]]e\ s$
    $= \{$I$=[[\pi \vdash$U:$\theta]]e\ s$.
  - $[[\pi \vdash$ **invoke** I: $\theta]]e\ s = v$,
    where (I$=v$) $\in e$.
- Example:
  - $[[\pi \vdash$ **const** I=E: $\{$I:τ*exp*$\}]]e\ s$
    $= \{$I$=[[\pi \vdash$ E: τ*exp*$]]e\ s\}$.
  - $[[\pi \vdash$ I: τ*exp*$]]e\ s = v$,
    where (I$=v$) $\in e$.
- Improper meaning for body:
  - $[[\pi \vdash$ **loop**: *intexp*$]]e\ s = \perp$.
  - $[[\pi \vdash$ **fun** A=**loop in skip**: *comm*$]]e\ s$
    $= [[\ \{A$: *intexp*$\} \vdash$ **skip**: *comm*$]]e_0\ s$,
    where $e_0 = \{A=[[\emptyset\vdash$ **loop**: *intexp*$]]\emptyset\ s\}$
    $= \{A = \perp$
    $= [[\{A$: *intexp*$\} \vdash$ **skip**: *comm*$]]\perp\ s$
    $= \perp$.

Other Standard Abstractions

P ::= D **in** C
D ::= **fun** I=E | **proc** I=C | **const** I=N | **alias** I=L
 | D_1,D_2 | D_1;D_2
C ::= L:=E | C_1;C_2 | **if** E **then** C_1 **else** C_2 **fi**
 | **while** E **do** C **od** | **skip** | **call** I
E ::= N | @L | E_1+E_2 | E_1=E_2 | ¬E | I
L ::= loc_i | I
N ::= n | I

Example:

- (**const** $A = 1$, **alias** X=loc_i);
  fun F = @X+A; **proc** P = X:=F;
  in call P; X:=A; **call** P.

Semantics

- General typing patterns:
  - $\dfrac{\pi \vdash \mathsf{U}{:}\theta}{\textbf{define } \mathsf{I}{=}\mathsf{U}{:}\ \{\mathsf{I}{:}\theta\}\textit{dec}}$
  - $\pi \vdash$**invoke** I: θ, if (I:θ) $\in \pi$
- Procedures: lazy evaluation.
  - $[[\pi \vdash$ **proc** I=C: $\{$I:*comm*$\}$*dec*$]]e\ s = \{$I$=r\}$,
    where $r(s') = [[\pi \vdash$ C: *comm*$]]e\ s'$.
  - $[[\pi \vdash$ **call** I: *comm*$]]e\ s = r(s)$
    where (I$=r$) $\in e$.
- Eager procedures: backtracking (non-local jumps).
  - $[[\pi \vdash$ **proc** I=C: $\{$I:*comm*$\}$*dec*$]]e\ s = \{$I$=s'\}$,
    where $s' = [[\pi \vdash$ C: *comm*$]]e\ s$.
  - $[[\pi \vdash$ **call** I: *comm*$]]e\ s = s'$, where (I$=s'$) $\in e$.
- Constants: eager evaluation.
  - $[[\pi \vdash$ **const** I=N: $\{$I:*int*$\}$*dec*$]]e\ s = \{$I$=n\}$,
    where $n = [[\pi \vdash$ N: *int*$]]e$.

Recursively Defined Abstractions

- Typing rule:
  - $$\frac{\pi \mathbin{\dot\cup} \{\mathsf{I}{:}\mathit{comm}\} \vdash \mathsf{C}{:}\ \mathit{comm}}{\textbf{rec-proc}\ \mathsf{I}{=}\mathsf{C}{:}\ \{\mathsf{I}{:}\mathit{comm}\}\mathit{dec}}$$
  - Procedure binding visible to own body!
- Repetitive constructs:
  - **while** E **do** C **od** = **call** W
    where **rec-proc** W = **if** E **then** C; **call** W **else skip fi**.
- Semantics:
  - $[[\pi \vdash$ **rec-proc** I=C: $\{$I:*comm*$\}$*dec*$]]e\ s$
    $= \{$I$=r\}$
    where $r(s') = [[\pi \mathbin{\dot\cup} \{$I:*comm*$\} \vdash$ C:*comm*$]]$
    $(e \cup \{$I$=r\})s'$.
- Copy rule:
  - **rec-proc** I=C = **define** I=(**rec** I:*comm*. C).
  - (**rec** I:*comm*.C) $\Rightarrow$ [(**rec** I:*comm*.C)/I]C.

Example

(**alias** A=loc_1, **alias** B=loc_2);
proc ADD = (**rec** ADD: *comm*.
if @A=0 **then skip else**
B:=@B+1; A:=@A-1; **call** ADD **fi**)
in A:=2; B:=3; **call** ADD

⇒ **alias** A=loc_1, **alias** B=loc_2,
proc ADD = (**rec** ADD: *comm*.
if @loc_1=0 **then skip else**
loc_2:=@loc_2+1; loc_1:=@loc_1-1; **call** ADD **fi**)
in A:=2; B:=3; **call** ADD

⇒ loc_1:=2; loc_2:=3;
(**rec** ADD: *comm*.
if @loc_1=0 **then skip else**
loc_2:=@loc_2+1; loc_1:=@loc_1-1; **call** ADD **fi**)

⇒ loc_1:=2; loc_2:=3;
if @loc_1=0 **then skip else**
loc_2:=@loc_2+1; loc_1:=@loc_1-1; (**rec** ADD: ... **fi**) **fi**)