The Parameterization and Correspondence Principles

Wolfgang Schreiner
Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

The Parameterization Principle

Phrases from any semantically meaningful syntactic class may be parameters.

- Potential parameters:
  - Numerals, expressions, locations, declarations, commands, type-structures, ...
- Potential constructs for parameterization:
  - For the moment, restrict to abstractions.
  - Applications of abstractions in variety of situations.

Parameterized Abstractions

- Parameterized V-abstraction:
  - **define** $\mathsf{I}_1(\mathsf{I}_2{:}\theta) = \mathsf{V}$
  - **invoke** $\mathsf{I}_1(\mathsf{U})$
- Parameters:
  - *Formal* parameter I_2
  - *Actual* parameter U.
- Typing of formal parameters:
  - Type attribute θ instead of type-structure value T.
  - Type-structure domain only describes *variables*.
  - May use *constant* values (integers, commands, etc) as parameters.

Typing Rules

- Abstraction:
  - $\dfrac{\pi \,\overline{\cup}\, \{\mathsf{I}_2{:}\theta_1\} \vdash \mathsf{V}{:}\theta_2}{\pi \vdash \textbf{define } \mathsf{I}_1(\mathsf{I}_2{:}\theta_1) = \mathsf{V}{:}\ \{\mathsf{I}_1{:}\ \theta_1 \rightarrow \theta_2\ \}\mathit{dec}}$
  - $\pi_1 \,\overline{\cup}\, \pi_2 = \pi_2 \,\overline{\cup} (\pi_1 - \{\ (\mathsf{I}{:}v) \mid (\mathsf{I}{:}w) \in \pi_2,\ (\mathsf{I}{:}v) \in \pi_1\})$.
  - Formal parameter cancels global identifier of same name.
- Invocation:
  - $\dfrac{\pi \vdash \mathsf{U}{:}\ \theta_1}{\pi \vdash \textbf{invoke } \mathsf{I}(\mathsf{U}){:}\ \theta_2}$ if $(\mathsf{I}{:}\theta_1 \rightarrow \theta_2) \in \pi$
  - Type of formal parameter must match type of actual parameter.
- Formal parameter reference:
  - $\pi \vdash \mathsf{I}{:}\ \theta$ if $(\mathsf{I}{:}\theta) \in \pi$
  - Formal parameters are referenced like abstractions (correspondence)!

Parameter Forms

Command abstractions

- Abstraction:
  - $\dfrac{\pi \mathbin{\overline{\cup}} \{\mathsf{I}_2{:}\theta_1\} \vdash \mathsf{C}{:}\ \mathit{comm}}{\pi \vdash \textbf{proc}\ \mathsf{I}_1(\mathsf{I}_2{:}\theta) = \mathsf{C}{:}\ \{\mathsf{I}_1{:}\ \theta \rightarrow \mathit{comm}\}\mathit{dec}}$
- Invokation:
  - $\dfrac{\pi \vdash \mathsf{U}{:}\ \theta}{\pi \vdash \textbf{call}\ \mathsf{I}(\mathsf{U}){:}\ \mathit{comm}}$ if $(\mathsf{I}{:}\theta\rightarrow \mathit{comm}) \in \pi$
- Other forms similar.

All *varieties of abstractions are legible for parameters!*

Expression Parameters

- E-phrases as parameters.
- θ= τ*exp*, $\tau \in \{$*int*, *bool*$\}$.
- Abstract syntax:
  - D ::= ...| **proc** $I_1(I_2$: τ*exp*) = C
    C ::= ...| **call** I(E)
    E ::= ...| I
- Notice:
  - **var** *A*: **newint**
    proc *P*(*M*: *intexp*) = *A*:=*M* **in call** *P*(@*A*)
  - **var** *A*: **newint**
    proc *P*(*M*: *intloc*) = *A*:=*M* **in call** *P*(@*A*)
  - *value* and *reference* parameters!

Evaluation Strategy

Crucial issue of parameters

- *Call-by-value*
  - Evaluate actual parameter immediately at point of invocation (*eager* evaluation).
- *Call-by-name*
  - Delay evaluation of actual parameter until the point of its use in the body of the abstraction (*lazy* evaluation).
- Example:
  - **var** A: **newint**
    proc $P(M\text{: } intexp) = A\text{:=}M;\ A\text{:=}M$
    in call $P(@A+1)$
  - Eager evaluation yields A=2.
  - Lazy evaluation yields A=3!

Copy Rule

- Lazy evaluated parameters:
  - **define** $I_1(I_2\colon\ \theta) = U \ldots$ **in** $\ldots$ **invoke** $I_1(V) \ldots$
  - $\ldots\ldots$ **in** $\ldots [V/I_2]U \ldots$
- Lazy evaluation difficult to understand when
  - Store changes as procedure evaluates.
  - Location is aliased by different names.

Lazy evaluation works better within functional languages!

Semantics of Parameter Transmission

- Parameterized procedure abstractions:
  - Lazy evaluated procedures.
  - Eagerly evaluated parameters.
  - $[[\mathsf{I}_1]]{=}p \in \mathsf{Env}_\pi \rightarrow [[\tau]] \rightarrow [[\mathit{comm}]]$
- Abstraction:
  - $[[\pi \vdash \textbf{proc}\ \mathsf{I}_1(\mathsf{I}_2\text{: } \tau\mathit{exp}) = \mathsf{C}\text{:}$
    $\{\mathsf{I}_1\text{: } \tau\mathit{exp} \rightarrow \mathit{comm}\}\mathit{dec}]]e\ s$
    $= (\{\mathsf{I}_1{=}p\},\ s)$
    where $p\ v\ s' = [[\pi\ \overline{\cup}\ \{\mathsf{I}_2\text{: } \tau\mathit{exp}\} \vdash \mathsf{C}\text{: } \mathit{comm}]](e\ \overline{\cup}\ \{\mathsf{I}_2{=}v\})\ s'$
  - $e\ \overline{\cup}\ \{\mathsf{I}{=}u\} = \{\mathsf{I}{=}u\} \cup (e \text{ - } \{(\mathsf{I}{=}v) \mid (\mathsf{I}{=}v) \in e\})$

Eagerly Evaluated Parameters

- Invocation:
  - $[[\pi \vdash$ **call** $\mathsf{I}_1(\mathsf{E})$: *comm*$]]e\ s$
    $= p([[\pi \vdash \mathsf{E}$: τ*exp*$]]e\ s)\ s$
    where $(\mathsf{I}_1{=}p) \in e$
- Formal parameter reference:
  - $[[\pi \vdash \mathsf{I}_2$: τ*exp*$]]e\ s = v$
    where $(\mathsf{I}_2{=}v) \in e$
- Problem: functions as parameters
  - Functions are lazily evaluated.
  - Semantics of identifer lookup depends on identifier type!
  - I: τ*exp* for function names.
  - I: τ*param* for formal parameters.
- Non-terminating parameter invocation:
  - $[[\pi \vdash$ **loop**: *intexp*$]]e\ s = \bot$
  - $[[\pi \vdash$ **call** $\mathsf{I}_1($**loop**$)$: *comm*$]]e\ s = p\ \bot\ s = \bot$.
  - p is *strict* on its actual parameter.

Lazy Evaluation

- Actual parameter evaluated with store only when corresponding formal parameter is referenced.
- Abstraction:
  - $[\![\pi \vdash$ **proc** $\mathsf{I}_1(\mathsf{I}_2\text{: } \tau\textit{exp}) = \mathsf{C}\text{:}$
    $\{\mathsf{I}_1\text{: } \tau\textit{exp} \rightarrow \textit{comm}\}\textit{dec}]\!]e\ s$
    $= (\{\mathsf{I}_1{=}p\},\ s)$
    where $p\ f\ s' = [\![\pi \,\overline{\cup}\, \{\mathsf{I}_2\text{: } \tau\textit{exp}\} \vdash \mathsf{C}\text{: } \textit{comm}]\!](e$
    $\overline{\cup}\ \{\mathsf{I}_2{=}f\})\ s'$

Lazy Evaluation

- Invocation:
  - $[[\pi \vdash$ **call** $\mathsf{I}_1(\mathsf{E})$: *comm*$]]e\ s = p\ f\ s$
    where $(\mathsf{I}_1{=}p) \in e$
    and $f s' = [[\pi \vdash \mathsf{E}$: τ*exp*$]]e\ s'$
- Formal parameter reference:
  - $[[\pi \vdash \mathsf{I}_2$: τ*exp*$]]e\ s = f(s)$
    where $(\mathsf{I}_2{=}f) \in e$
  - Current store is supplied to formal parameter!

Soundness of Typing Rules

- Type attributes
  - $\theta ::= \tau \mid \tau\,\mathit{exp} \mid \mathit{comm} \mid \delta \mid \delta\,\mathit{class} \mid \pi\,\mathit{dec}$
    $\mid \theta_1 \rightarrow \theta_2$.
  - $\tau ::= \mathit{int} \mid \mathit{bool}$
  - $\delta ::= \mathit{intloc} \mid \pi$
  - $\pi ::= \{j{:}\theta_j\}_{j \in J}$, $\mathsf{J} \subseteq$ Identifier.
- Lazy evaluation semantics:
  - $[[\theta_1 \rightarrow \theta_2]] = [[\theta_1]] \rightarrow [[\theta_2]]$
- Prove soundness of typing rules:
  - $[[\pi \vdash \mathsf{U}{:}\theta]] \in \mathsf{Env}_\pi \rightarrow [[\theta]]$
  - $[[\pi \vdash$ **define** $\mathsf{I}_1(\mathsf{I}_2{:}\theta_1) = \mathsf{U}{:}\ \{\mathsf{I}{:}\theta_2\}\mathit{dec}]]e \in$
    $[[\{\mathsf{I}_1{:}\theta_1 \rightarrow \theta_2\}\mathit{dec}]]$
  - $[[\pi \cup \{\mathsf{I}_2{:}\ \theta_1\} \vdash \mathsf{U}{:}\theta_2]](e \cup \{\mathsf{I}_2{=}v\}) \in [[\theta_2]]$
    for every $v \in [[\theta_1]]$
- Eager evaluation semantics more complex:
  - $[[\tau\,\mathit{exp} \rightarrow \mathit{comm}]] = [[\tau]] \rightarrow [[\mathit{comm}]]$

Copy Rule

- Lazy evaluated parameters:
  - **define** $\mathsf{I}_1(\mathsf{I}_2\colon\ \theta) = \mathsf{U}$ ...**in** ...**invoke** $\mathsf{I}_1(\mathsf{V})$ ...
  - ......**in** ...$[\mathsf{V}/\mathsf{I}_2]\mathsf{U}$ ...
- Augment copy rule by rule for parameter binding.
- *Lambda abstraction*
  - **define** $\mathsf{I}_1 = \lambda \mathsf{I}_2{:}\theta.\mathsf{U}$
  - $\dfrac{\pi \mathbin{\overline{\cup}} \{\mathsf{I}{:}\theta_1\} \vdash \mathsf{E}\colon\ \theta_2}{\pi \vdash (\lambda \mathsf{I}{:}\theta_1.\mathsf{E})\colon\ \theta_1 \to \theta_2}$
  - $[[\pi \vdash \lambda \mathsf{I}{:}\theta_1.\mathsf{E}\colon\ \theta_1 \to \theta_2]]e = f$
    where $f\ v = [[\pi \mathbin{\overline{\cup}} \{\mathsf{I}{:}\theta_1\} \vdash \mathsf{E}\colon\ \theta_2]](e \mathbin{\overline{\cup}} \{\mathsf{I}{=}v\})$

Parameter Copy Rule

- Example:
  - **var** A: **newint**; **proc** $P = \lambda X$: *intexp*.A:=X **in call** P(@A+1)
  - **var** A: **newint in** (λX: *intexp*.A:=X)(@A+1)
- Rule for binding @A+1 to X:
  - (λI:θ.U)V $\Rightarrow$ [V/I]U
  - **var** A: **newint in** [@A+1/X](A:=X) = **var** A: **newint in** A:=@A+1
- Dangerous: substitution into lambda abstractions.
  - [V/I](λI'.U)
  - **var** A: **newint**; **fun** F = @A+1; **fun** G = (λA:*intexp*.A+F) **in** ...

Parameter Substitution

- Copy rule for sequential declarations
  - **define** I=U; D
    $\Rightarrow$ **define** I=U, [U/I]D
- Transformation sequence:
  - $\Rightarrow$ **var** A: **newint**; (**fun** F = @A+1,
    fun G = [@A+1/F](λA:*intexp*.A+F)) **in** ...
  - $\Rightarrow$ **var** A: **newint**; (**fun** F = @A+1,
    fun G = (λA:*intexp*. A+(@A+1))) **in** ...
- Rename locally defined identifiers:
  - $\Rightarrow$**var** A: **newint**; (**fun** F = @A+1,
    fun G = [@A+1/F](λB:*intexp*.B+F)) **in** ...
  - $\Rightarrow$**var** A: **newint**; (**fun** F = @A+1,
    fun G = (λB:*intexp*. B+(@A+1))) **in** ...

Numeral and Command Parameters

- Example:
  - **const** K=2; **var** A; **newint**;
    proc P(M: *comm*) = A:=K+1; M;
    proc Q(X: *int*) = A:=X
    in call P(**call** Q(2));
    call P(**call** P(**call** Q(K)));
    call P(A:=K)
- Command parameters evaluated lazily.
- Eagerly evaluated command parameters:
  - Argument defines *backtracking point*.
  - [[π ⊢**call** I(C): *comm*e s
    = p([[π ⊢ C: *comm*]]e s)s
    where (I=p) $\in e$
  - **var** A; **newint**;
    proc P(M: *comm*) =
    ...**if** *error* **then** M **else** ...**fi**
    in ...**call** P(A:=0)
  - Reference of M extracts old store!

Declaration and Variable Parameters

- Declaration parameters:
  - **module** *COUNTER*
    (M: $\{X$:*intloc*, *INIT*:*comm*$\}$*dec*) =
    $\{$**import** M;
    proc P = (**call** *INIT*; X:=@X+1)$\}$;
    module N = $\{$**var** X: **newint**;
    proc *INIT* = X:=0$\}$;
    import *COUNTER*(**import** N) **in** ...
- Variable parameters:
  - **class** K = **record**
    var A: *newint*
    proc P(B: *intloc*) = A:=@A+@B
    end;
    var R: K;
    proc Q(Y: $\{A$: *intloc*, P: *intloc* $\rightarrow$ *comm*$\}$)
    = **call** $Y.P(Y.A)$
    in A:=0; **call** $Q(R)$
  - Type structures are storage allocation routines and *not* type attributes!

Type Equivalence

Type attributes must match.

- Problem with type structures:
  - **class** R = **record var** X: **int**, **var** Y: **int end**;
    var E: R,
    var F: **record var** X: **int**, **var** Y: **int end**,
    var G: **record var** Y: **int**, **var** X: **int end**,
    proc $P(Q\colon \{X\colon \mathit{intloc}, Y\colon \mathit{intloc}\}) = \ldots$
- *Structural type equivalence*:
  - All have same type attribtute.
  - E, F, G can bind to Q.
  - Current typing rules.
- *Name (occurence) equivalence*:
  - Each class definition names *new* type.
  - Type structures used in place of type attributes.
  - **proc** $P(Q\colon R) = \ldots$
  - Only E can bind to Q!

Name Equivalence

- Complicated in practice:
  - Primitive types (*intloc*) are always treated equivalent (named by class definition or not).
  - Each occurence of compound type structures (**array**, **record**) is treated as new type (whether named or not).
- Supports "encapsulation"
  - Procedures parameterized on one type of record structure cannot be abused by giving another.
  - **record** x: **newreal**; y: **newreal end**
    record r: **newreal**; phi: **newreal end**
- Straight-forward to implement.
  - Attach unique integer tag to each compound type-structure expression.
- Less natural logic.
  - No basic problem.
  - Just fewer programs well-typed.

Type-Structure Parameters

- Build different versions of classes:
  - **class** *STACK1* = **record var** *A*: **newint**;
    proc *PUSH* = *A*:=0 **end**
    class *STACK2* = **record var** *A*: **newint**;
    proc *PUSH* = *A*:=1 **end**
    class *K*(*T*: {*A*: *intloc*, *PUSH*: *comm*}*class*) =
    record var *S*:*T*; **proc** *PUSH2* =
    call *S.PUSH*; **call** *S.PUSH* **end**;
    var *X*: *K*(*STACK1*); **var** *Y*: *K*(*STACK2*)
  - *K* is *generic* in the data structure it handles.
- Type-structure parameters are lazily evaluated
  - Eager evaluation would activate type structure and bind allocated storage to formal parameter.

Generalization

- Parameters with unknown type attributes (Ada)
  - **module** $M(T$: Type-structure)
    $=\{$**var** $A{:}T$; **proc** $SAVE(X{:}T) = A{:=}@X\}$
    …**import** M(**newint**) …**in call** $SAVE(A)$
  - All uses of T in M independent of actual value of T.
- Typing rules:
  - M: $\delta class \rightarrow \{\mathrm{A}{:}\delta,\ SAVE{:}\delta\rightarrow\ comm\}dec$
  - **import** M(**newint**) instantiates δ by *intloc*
  - **import** M(**newint**):
    $\{\mathrm{A}{:}intloc,\ SAVE{:}intloc \rightarrow comm\}dec$

Parametric Polymorphism.

Denotational Semantics

- Type attributes become values in the semantics.
  - $[[\pi \vdash$ **module** $\mathsf{I}_1(\mathsf{I}_2$:Type-structure$) =$
    D:$\{\mathsf{I}_1$:δ*class* $\to \theta(\delta)\}$*dec*$]]e\ s$
    $=(\{\mathsf{I}_1{=}m\},\ s)$
    where $m(\delta$*class*$)\ v\ s'$
    $= [[\pi \cup \{\mathsf{I}_2$:$\delta$*class*$\} \vdash$ D: $\theta(\delta)]](e \cup \{\mathsf{I}_2{=}v\})s'$
  - $[[\pi \vdash$ **import** I(T):$\theta(\delta)]]e\ s$
    $= m(\delta$*class*$)([[\pi \vdash$ T: δ*class*$]]e)s$
    where $(\mathsf{I}{=}m) \in e$
  - $m \in \delta$*class* $\to [[\delta$*class*$]] \to$ *Store*
    $\to ([[\theta(\delta)]] \times$ *Store*$)$
- Argument structure depends on m's first argument!
- *Dependent types*

Serious questions about soundness open.

The Correspondence Principle

$\mathsf{D} \in$ Declaration
$\mathsf{U} \in$ Any-syntax-domain
$\mathsf{A} \in$ Actual-parameter

$\mathsf{D} ::= \ldots |$ **define** $\mathsf{I}_1(\mathsf{I}_2\!: \theta) = \mathsf{U}$
$\mathsf{U} ::= \ldots |$ **invoke** $\mathsf{I}(\mathsf{A})$
$\mathsf{A} ::= \mathsf{N} \mid \mathsf{E} \mid \mathsf{C} \mid \mathsf{L} \mid \mathsf{D} \mid \mathsf{T}$

- Various parameters for U-abstractions.
- Criterium whether parameter forms fit to original language?
- *Correspondence Principle*
  - **define** $\mathsf{I}_2 = \mathsf{U}$.
  - **define** $\mathsf{I}_1(\mathsf{I}_2\!:\!\theta) = \ldots$ **in** $\ldots \mathsf{I}_1(\mathsf{U})$.
  - Semantics of binding I_2 to U is the same.

For every abstraction form there may be a corresponding parameterization form (and vice versa) with same binding semantics.

Consequences of Correspondence

- Semantics of identifiers:
  - Independent of way in which identifier is bound.
  - Understanding more easy.
- Direction for language extension:
  - Existing abstraction (e.g. procedures).
  - Introduce parameter form (e.g. command parameter).
- Correspondence may fail:
  - Evaluate functions lazily but expression parameters eagerly.
  - **var** A: **newint**; **fun** $F = @A+1$
    proc $P(X\text{: } \mathit{intexp}) = A{:=}X+F;\ A{:=}X+F$
  - Both X and F have type *intexp*.
  - X and F are evaluated differently.
  - Syntactic distinction: $A{:=}X+(\textbf{eval } F)$

Consequences of Correspondence

- Presence of both binding forms:
  - Numeral abstractions but not parameters.
  - Numeral parameters are *static parameters*.
  - Compiler may generate *specialized* abstractions.
  - **static-fun** $F(X\text{: }int) = X{+}1$
    var A: **newint in** $A{:=}F(1){+}F(3)$
  - **var** A: **newint in** $A{:=}2{+}4$
- Combination of static/dynamic parameters:
  - **proc** $SORT(L\text{: }int,\ V\text{: }list\text{-}of\ int)$
    $= \ldots L \ldots V \ldots$
    in … **call** $SORT(3,\ [4{,}2{,}6])$
    … **call** $SORT(4{,}[0{,}1{,}3{,}2)$ …
  - **proc** $SORT3(V\text{: }list\text{-}of\ int) = \ldots 3 \ldots V \ldots$
    proc $SORT4(V\text{: }list\text{-}of\ int) = \ldots 4 \ldots V \ldots$
    in … **call** $SORT3([4{,}2{,}6])$
    … **call** $SORT4([0{,}1{,}3{,}2])$

Theory of partial evaluation!

Semantics of Correspondence

- Lazy evaluation semantics of functions:
  - $[[\pi \vdash$ **fun** I_2=E: $\{$I: τ*exp*$\}$*dec*$]]e\ s$
    $= (\{\mathsf{I}_2 = f\},\ s)$
    where $f(s') = [[\pi \vdash$ E: τ*exp*$]]e\ s'$
  - $[[\pi \vdash \mathsf{I}_2$: τ*exp*$]]e\ s' = f(s')$
    where $(\mathsf{I}_2 = f) \in e$
- Semantics of expression parameters:
  - $[[\pi \vdash$ **define** $\mathsf{I}_1(\mathsf{I}_2$: τ*exp*$) =$
    U: $\{\mathsf{I}_1$: τ*exp* $\rightarrow \theta\}$*dec*$]]e\ s$
    $= (\{\mathsf{I}_1 = g\},\ s)$
    where $g\ f\ s'$
    $= [[\pi \cup \{\mathsf{I}_2$:$\tau$*exp*$\} \vdash$ U: $\theta]](e \cup \{\mathsf{I}_2 = f\})$
    s'
  - $[[\pi \vdash$ **invoke** I_1(E): $\theta]]e\ s = g\ f\ s$
    where $(\mathsf{I}_1 = g) \in e$
    and $f\ s' = [[\pi \vdash$ E: τ*exp*$]]e\ s'$
  - $[[\pi \vdash \mathsf{I}_2$: τ*exp*$]]e\ s' = f(s')$
    where $(\mathsf{I}_2 = f) \in$ *e*

Proof of Correspondence

- Equality of meanings:
  - $[\![$**fun** $\mathsf{I}_2 = \mathsf{E}$ **in** $\ldots$: $\theta]\!]e\ s$
    $= [\![$**define** $\mathsf{I}_1(\mathsf{I}_2{:}\tau\mathit{exp}) = \ldots$
    in invoke $\mathsf{I}_1(\mathsf{E})$: $\theta]\!]e\ s$

- $[\![$**fun** $\mathsf{I}_2 = \mathsf{E}$ **in** $\ldots$: $\theta]\!]e\ s$
  $= [\![\pi_2 \vdash \ldots: \theta]\!](e \,\overline{\cup}\, \{\mathsf{I}_2{=}[\![\pi \vdash \mathsf{E}: \tau\mathit{exp}]\!]e\})\ s$
  where $\pi_2 = \pi \,\overline{\cup}\, \{\mathsf{I}_2{:}\tau\mathit{exp}\}$

- $[\![$**define** $\mathsf{I}_1(\mathsf{I}_2{:}\tau\mathit{exp}) = \ldots$**in invoke** $\mathsf{I}_1(\mathsf{E})$: $\theta]\!]e\ s$
  $= [\![\pi \vdash$ **invoke** $\mathsf{I}_1(\mathsf{E})$: $\theta]\!]e_1\ s$
  where $\pi_1 = \pi \,\overline{\cup}\, \{\mathsf{I}_1{:}\ \tau\mathit{exp} \to \theta\}$,
  and $\mathsf{e}_1 = e \,\overline{\cup}\, \{\mathsf{I}_1{=}g\}$
  and $g\ f\ s' = [\![\pi_2 \,\overline{\cup}\, \ldots: \theta]\!](e \,\overline{\cup}\, \{\mathsf{I}_2{=}f\})\ s'$
  $= g\ f_1\ s$ where $f_1\ s' = [\![\pi_1 \vdash \mathsf{E}: \tau\mathit{exp}]\!]e_1\ s'$
  $= [\![\pi_2 \vdash \ldots: \theta]\!](e \,\overline{\cup}\, \{\mathsf{I}_2{=}f_1\})\ s$
  $= [\![\pi_2 \vdash \ldots: \theta]\!](e \,\overline{\cup}\, \{\mathsf{I}_2{=}f_2\})\ s$
  where $f_2\ s' = [\![\pi \vdash \mathsf{E}: \tau\mathit{exp}]\!]e\ s'$
  (since I_1 does not appear in E)

Proof adapts to eager evaluation as well.

Parameter Lists

- Bind multiple values to multiple names.
  - **define** I(I$_1$: θ_1, I$_2$: θ_2, $\ldots$, I$_n$: θ_n) = U
  - **invoke** I(V$_1$, V$_2$, $\ldots$, V$_n$)
- Sample syntax:
  - D ::= $\ldots$| **define** I(FL)=U
    FL ::= I:θ| I:θ, FL
    U ::= $\ldots$| **invoke** I(AL)
    AL ::= A | A, AL
  - A bit cumbersome for writing type rules.
- Correspondence principle:
  - Binding of multiple parameters.
  - Multiple definition bindings (compound declaration)!
- Alternative syntax:
  - D ::= $\ldots$| **define** I(F)=U
    F ::= I:θ| F$_1$, F$_2$
    V ::= $\ldots$| **invoke** I(D)

Typing of Parameter Lists

- Typing rules:
  - $\dfrac{\pi \,\overline{\cup}\, \pi_1 \vdash \mathsf{U}\colon \theta}{\pi \vdash \textbf{define } \mathsf{I(F)} = \mathsf{U}\colon \{\mathsf{I}\colon \pi_1 \to \theta\}\mathit{dec}}$
  - $\dfrac{\pi \vdash \mathsf{D}\colon \pi_1\mathit{dec}}{\pi \vdash \textbf{invoke } \mathsf{I(D)}\colon \theta}$ if $(\mathsf{I}\colon \pi_1 \to \theta) \in \pi$
  - $\pi_1 = \mathit{type\text{-}attrs}(\mathsf{F})$
    $\mathit{type\text{-}attrs}(\mathsf{I}{:}\theta) = \{\mathsf{I}{:}\theta\}$
    $\mathit{type\text{-}attrs}(\mathsf{F}_1, \mathsf{F}_2) =$
    $\quad \mathit{type\text{-}attrs}(\mathsf{F}_1) \,\dot{\cup}\, \mathit{type\text{-}attrs}(\mathsf{F}_2)$
- Example:
  - **var** *A*: **newint**;
    proc *P*(*W*:*intexp*, *X*:*intloc*, *Y*:*comm*,
    Z:*comm* → *comm*)
    = *X*:=*W*; **call** *Z*(*Y*);
    proc *Q* = *A*:=0
    in call *P*(**fun** *W*=@*A*+2, **proc** *Y* = **call** *Q*,
    alias *X*=*A*, **proc** *Z*(*C*:*comm*) = **call** *C*)

Labeled parameter passing (Ada).

Semantics of Parameter Lists

- Defined via semantics of *declarations*:
- Abstraction:
  - $[[\pi \vdash$ **define** I(F)=U: $\{$I:$\pi_1 \rightarrow \theta\}$*dec*$]]e\ s$
    $= (\{$I=$p\}, s)$
    where $p\ e_1\ s_1 =$
    $[[\pi \cup \pi_1 \vdash$ U: $\theta]](e \cup e_1)\ s_1$
    and $\pi_1 =$ *type-attrs*(F)
- Invocation:
  - $[[\pi \vdash$ **invoke** I(D): $\theta]]e\ s = p\ e_1\ s_1$
    where (e$_1$, s$_1$) $= [[\pi \vdash$ D: π_1*dec*$]]e\ s$
    and (I=p) $\in e$
  - Store at point of invocation is supplied to actual parameters.
  - Abstractions are evaluated lazily, store is ignored.
  - Abstractions are evaluated eagerly, store is used.

Correspondence is forced upon language!

Lambda Abstraction Principle

The parameterization principle is a lambda abstraction principle.

- Parameterized abstraction is just abstraction whose body is parameterized phrase.
- Lambda notation emphasizes this point.
  - λI:θ. U
- Parameterization is *distinct* from abstraction!
  - Parameterized phrases need not be named.
  - $(\lambda X$:*intexp*. A:=X)(@A+1)
- Typing rules use new type attribute.
  - $\theta_1 \rightarrow \theta_2$

Abstraction and parameterization are orthagonal principles.

Summary (Typing Rules)

- Abstract syntax:
  - D ::= ... | **define** $\mathsf{I}_1(\mathsf{I}_2{:}\ \theta) = \mathsf{U}$
    U ::= ... | **invoke** I(V)
    V ::= ... | I
- Typing rules:
  - $$\frac{\pi \mathbin{\overline{\cup}} \{\mathsf{I}_2{:}\theta_1\} \vdash \mathsf{U}{:}\theta_2}{\pi \vdash \textbf{define } \mathsf{I}_1(\mathsf{I}_2{:}\theta_1) = \mathsf{U}{:}\ \{\mathsf{I}_1{:}\ \theta_1 \rightarrow \theta_2\ \}\mathit{dec}}$$
  - $$\frac{\pi \vdash \mathsf{V}{:}\ \theta_1}{\pi \vdash \textbf{invoke } \mathsf{I}(\mathsf{V}){:}\ \theta_2} \quad \text{if } (\mathsf{I}{:}\theta_1 \rightarrow \theta_2) \in \pi$$
  - $\pi \vdash \mathsf{I}{:}\ \theta \quad$ if $(\mathsf{I}{:}\theta) \in \pi$
- Type attributes:
  - $\theta ::= \tau \mid \tau\mathit{exp} \mid \mathit{comm} \mid \delta \mid \delta\mathit{class} \mid \pi\mathit{dec}$
    $\mid \theta_1 \rightarrow \theta_2$.
  - $\tau ::= \mathit{int} \mid \mathit{bool}$
  - $\delta ::= \mathit{intloc} \mid \pi$
  - $\pi ::= \{j{:}\theta_j\}_{j \in J}$, J $\subseteq$ Identifier.

Summary (Semantics)

- Lazy evaluation semantics:
  - $[[\pi \vdash$ **define** $\mathsf{I}_1(\mathsf{I}_2\text{: } \tau\,exp) =$
    $\mathsf{U}\text{: } \{\mathsf{I}_1\text{: } \tau\,exp \to \theta\}dec]]e\ s$
    $= (\{\mathsf{I}_1{=}p\},\ s)$
    where $p\ f\ s = [[\pi \cup \{\mathsf{I}_2{:}\theta_1\} \vdash \mathsf{U}\text{: } \theta_2]](e \cup \{\mathsf{I}_2{=}f\})\ s$
  - $[[\pi \vdash$ **invoke** $\mathsf{I}_1(\mathsf{V})\text{: } \theta_2]]e\ s = p\ f\ s$
    where $(\mathsf{I}_1{=}p) \in e$
    and $f\ s' = [[\pi \vdash \mathsf{V}\text{: } \theta_1]]e\ s'$
  - $[[\pi \vdash \mathsf{I}_2\text{: } \theta]]e\ s = f(s)$
    where $(\mathsf{I}_2{=}f) \in e$

Summary (Lambda Abstraction)

- Abstract syntax:
  - D ::= ... | **define** I_1 = A
    A ::= λI:θ. U
    U ::= ... | **invoke** I(V)
    V ::= ... | I
- Typing rules:
  - $$\frac{\pi \uplus \{\mathsf{I}{:}\theta_1\} \vdash \mathsf{E}{:}\theta_2}{\pi \vdash (\lambda \mathsf{I}{:}\theta_1.\mathsf{E}){:}\ \theta_1 \to \theta_2}$$
- Semantics:
  - $[[\pi \vdash \lambda \mathsf{I}{:}\theta_1.\mathsf{E}{:}\ \theta_1 \to \theta_2]]e = f$
    where $f\ v = [[\pi \uplus \{\mathsf{I}{:}\theta_1\} \vdash \mathsf{E}{:}\theta_2]](e \uplus \{\mathsf{I}{=}v\})$
- Parameter copy rule:
  - $(\lambda \mathsf{I}{:}\theta.\ \mathsf{U})\mathsf{V} \Rightarrow [\mathsf{V}/\mathsf{I}]\mathsf{U}$