A Denotational Engineering of Programming Languages

…

Part 6: Lingua-2 – Procedures
(Section 6 of the book)

Andrzej Jacek Blikle

August 24th, 2022

Lingua-2 is an extension of Lingua-1

Lingua-2 emerges from Lingua-1 by extending the algebras of syntax and of denotations by:

- new carriers (e.g. procedure contents),
- new reachable elements in old carriers (e.g. procedure calls),
- new constructors.

Everything else remains unchanged

Values and states

Recapitulation

sta	: State	= Env x Store	
env	: Env	= TypeEnv x ProEnv	environments
sto	: Store	= Valuation x (Error | {'OK'})	
vat	: Valuation	= Identifier $\Rightarrow$ (Value | PsValue)	
tye	: TypEnv	= Identifier $\Rightarrow$ (Type | Body)	type environments
pre	: ProEnv	= Identifier $\Rightarrow$ Procedure	procedure environments

Procedure names

sta = ((tye, pre), (vat, err))   err : Error | {'OK'}

this structure is not accidental

Imperative procedures

Imperative procedures

pro : Procedure = ImpPro | FunPro
apd : AcPaDe = $Identifier^{c*}$ the denotations of actual parameters (val. and ref.)
ipr : ImpPro = AcPaDe x AcPaDe $\mapsto$ Store $\to$ Store imperative procedures

actual value-parameter denotations

actual reference-parameter denotations

stores used to avoid self-applicability, a mathematical decision

Procedures have no syntactic counterparts!

We do not talk about "denotations of procedures". They are denotations themselves!

At the level of syntax we only have:

- procedure declaration,
- procedure calls

actual parameters = identifiers (rather than arbitrary expressions)
an engineering decision which simplifies:

- proof rules
- recursion mechanism (expressions may include procedure calls)

A

Why procedures can't return state-to-state functions?

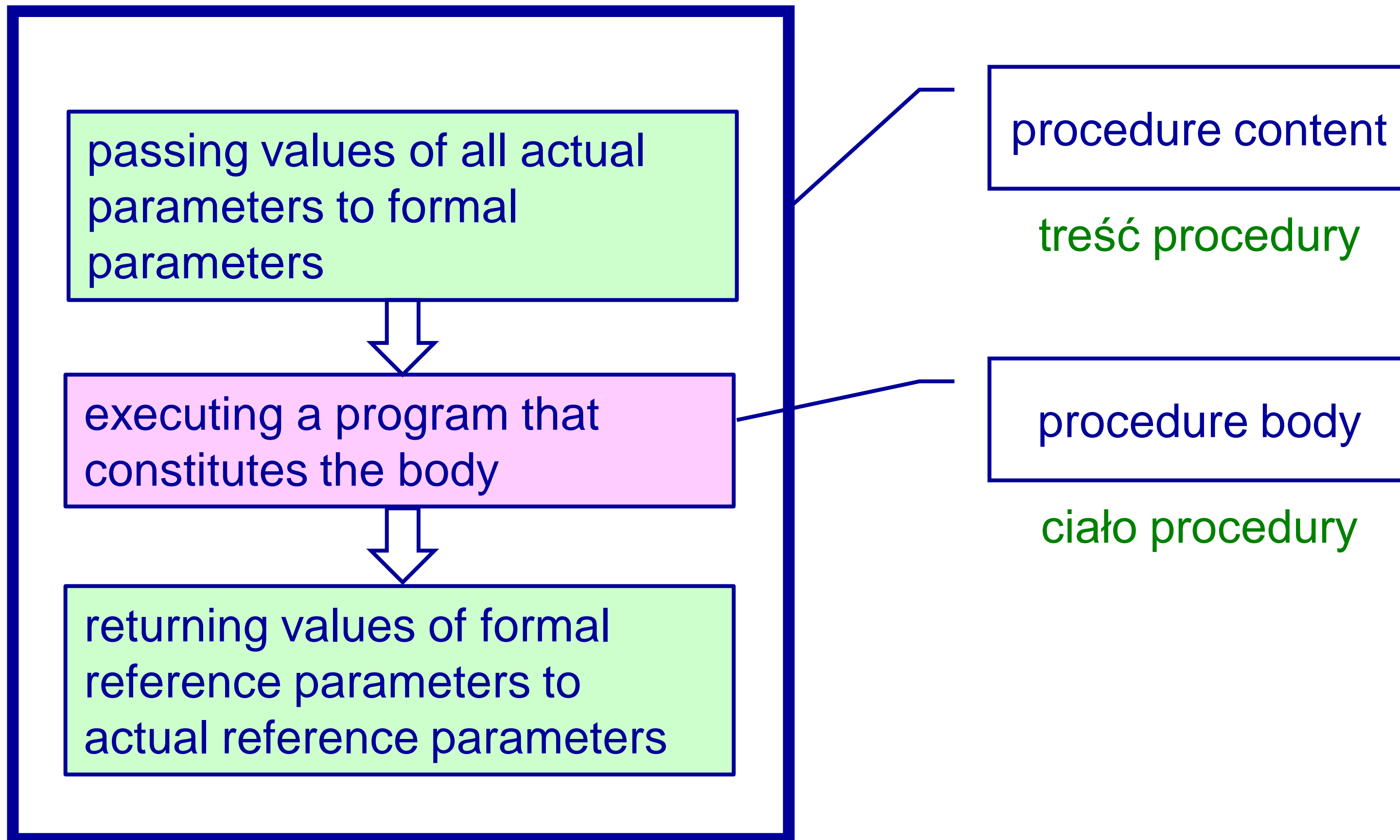
If they were:

ImpPro = AcPaDe x AcPaDe ⟼ State → State

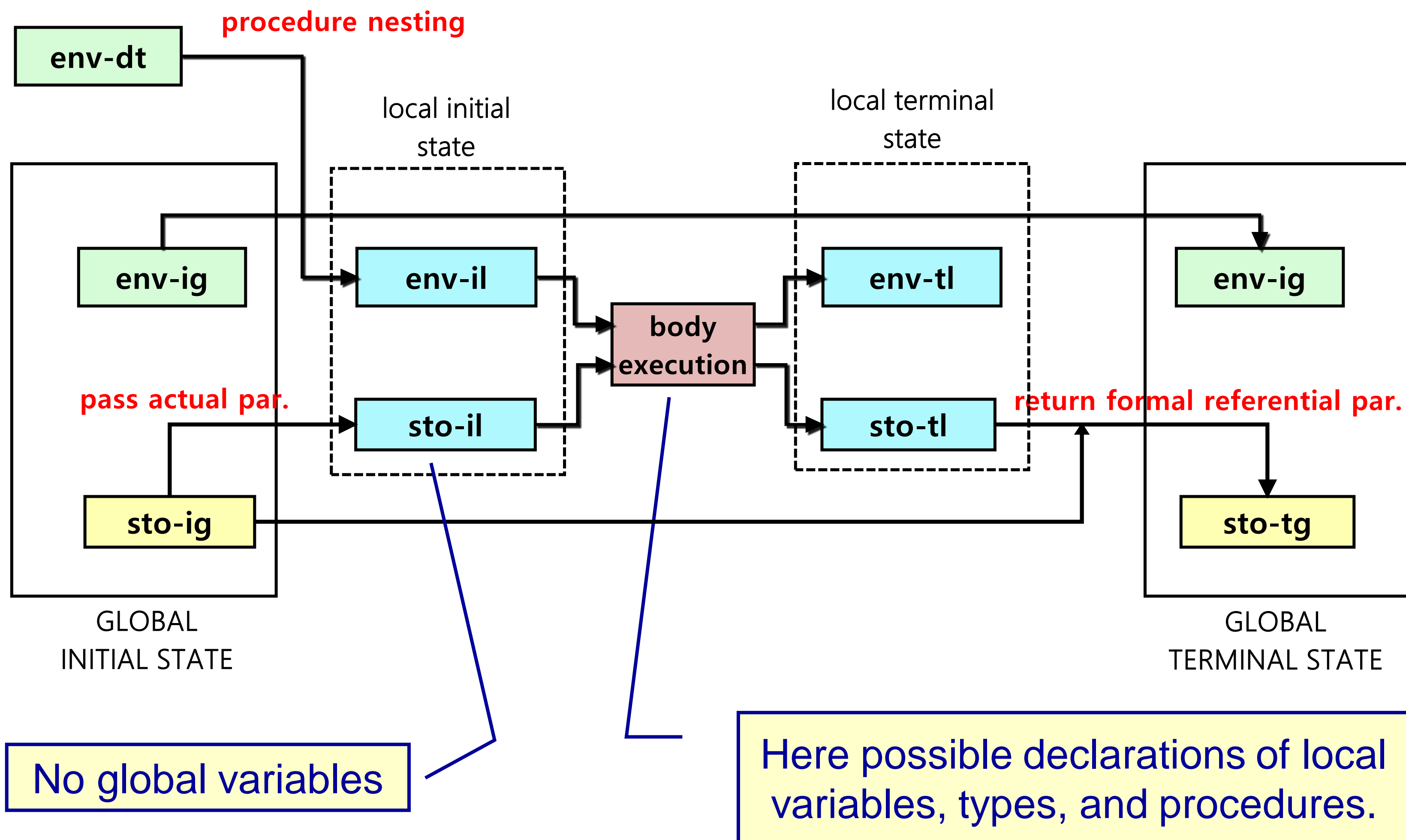
State = (TypEnv x ProEnv) x Store

ProEnv = Identifier ⟹ ImpPro

The execution of a procedure call



The execution of a procedure call



The communication of procedures with hosting programs

Data processing (values)

Input – actual parameters (value and reference)

Output – reference parameters

Access to procedures and types

Declaration-time environment

No global variables!

The denotations of actual parameters

apd : AcPaDe = Identifier^{c*} — actual-parameter denotations

ceate-empty-act-par-den : $\mapsto$ AcPaDe

create-empty-act-par-den.() = ()

create-single-act-par-den : Identifier $\mapsto$ AcPaDe

create-single-act-par-den.ide = (ide)

(a,b,c) ₵ (d) = (a,b,c,d)

add-act-par-den : AcPaDe x Identifier $\mapsto$ AcPaDe

add-act-par-den.(apd, ide) = apd ₵ (ide)

Actual parameters: value parameters or reference parameters.

The denotations of formal parameters

fpd : FoPaDe = (Identifier x TypExpDen)^{c*}

create-empty-for-par-den : ⟼ FoPaDe

create-empty-for-par-den.() = ()

create-single-for-par-den : Identifier x TypExpDen ⟼ AcPaDe

create-single-for-par-den.(ide, ted) = ((ide, ted))

add-for-par-den : AcPaDe x Identifier x TypExpDen ⟼ AcPaDe

add-for-par-den.(fpd, ide, ted) = apd ₵ (ide)

Formal parameters: value parameters or reference parameters.

Compatibility of formal with actual parameters

statically-compatible : FoPaDe x FoPaDe x AcPaDe x AcPaDe $\mapsto$ Error | {'OK'}

1. no repetitions on the combined list of formal parameters
2. no repetitions on the list of actual reference-parameters,
3. mutually corresponding lists of actual and formal par. (have the same length)

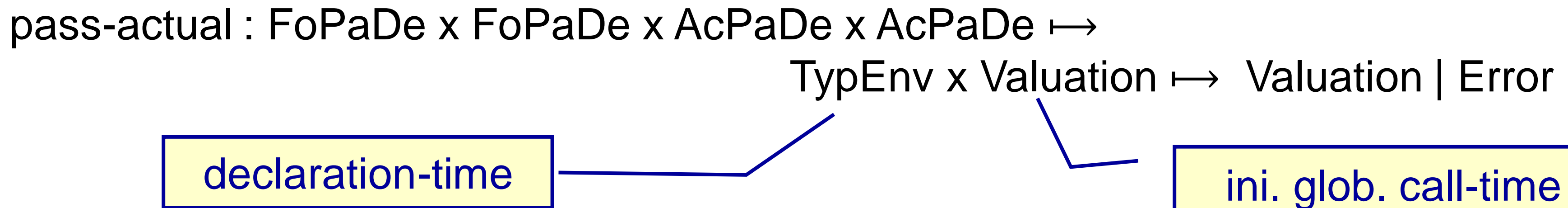
dynamically-compatible : FoPaDe x FoPaDe x AcPaDe x AcPaDe $\mapsto$
TypEnv x Valuation $\mapsto$ Error | {'OK'}

declaration-time environment

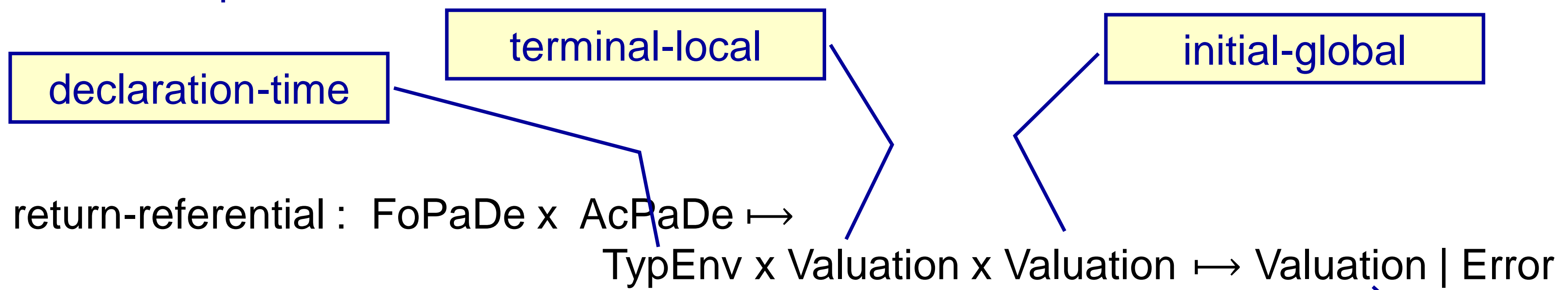
call-time valuation

1. statically compatible,
2. actual parameters declared (not necessarily initialized, e.g. ref. par.)
3. type expressions assigned to formal parameters evaluate to non-errors,
4. bodies of corresponding actual and formal parameters coincide,
5. composites of actual parameters satisfy the yokes of formal parameters.

Parameter passing



1. Checking dynamic compatibility of formal with actual parameters.
2. Values (or pseudovalues) of actual parameters are passed to corresponding formal parameters.



Values (or pseudovalues) of formal reference parameters and returned to corresponding actual param.

Constructor of imperative procedures (introduction)

The denotations of the contents (building blocks) of imperative-procedure

icd : IprConDen = FoPaDe x FoPaDe x ProDen

treść procedury   value   reference

program denotation (procedure body)

create-imp-con : FoPaDe x FoPaDe x ProDen $\mapsto$ IprConDen

create-imp-con.(fpd-1, fpd-2, prd) = (fpd-1, fpd-2, prd)   contents of imp. proc.

declaration-time environment

create-imp-proc : IprConDen x Env $\mapsto$ ImpPro   i.e.

create-imp-proc : (FoPaDe x FoPaDe x ProDen) x Env $\mapsto$

AcPaDe x AcPaDe $\mapsto$ Store $\rightarrow$ Store

Constructor of imperative procedures

declaration time

create-imp-proc.(fpd-v, fpd-r, prd).env-dt.(apd-v, apd-r).ct-sto =

is-error.ct-sto ➔ ct-sto
let
(ct-vat, ‘OK’) = ct-sto - initial global
(dt-tye, dt-pre) = dt-env - declaration time
par = (fpd-v, fpd-r, apd-v, apd-r)
il-vat = pass-actual.par.(dt-tye, il-vat) - call-time initial local
il-vat : Error ➔ (vat-ig, il-vat)
let
il-sta = (dt-env, (il-vat, 'OK')) - initial local
prd.il-sta = ? ➔ ?
let procedure-body execution
(tl-env, (tl-vat, err)) = prd.sta-il - terminal local
err ≠ 'OK' ➔ ct-sto ◄ err
let
tg-sta = return-referential.(fpd-r, apd-r).(dt-tye, vat-tl, vat-ig) - terminal global
is-error.tg-sta ➔ (ct-vat, error.tg-sta)
let
(tg-env, (tg-vat, 'OK')) = tg-sta
true ➔ (tg-vat, ‘OK’)

compatibility of formal with actual checked

call time store

Declaration of a procedure

declare-imp-pro : Identifier x IprConDen ⟼ IprDecDen   i.e.

declare-imp-pro : Identifier x FoPaDe x FoPaDe x ProDen ⟼ State ⟼ State

declare-imp-pro.(ide, fpd-v, fpd-r, prd).sta =
  is-error.sta   ➔ sta
  ide : declared.sta   ➔ sta ◄ 'identifier-declared'
  let
    ((tye, pre), (vat, err)) = sta
    ipr = create-imp-proc.(fpd-v, fpd-r, prd, (tye, pre[ide/ipr])) - fixed-point recursion
  true   ➔ ((tye, pre[ide/ipr]), (sto, 'OK'))

Declaration may be trivial (`skip-d`)

Recursion in **MetaSoft** permits to describe recursion in **Lingua** without a stack mechanizm.

A

Recursion – how it works?

ipr = create-imp-proc.(fpd-v, fpd-r, prd, (tye, pre[ide/ipr]))

Kleene's fixed-point theorem

If $f : A \mapsto A$ is continuous, then the least solution of

$x = f.x$

exists and equals $\lim(f^n.\Phi \mid n = 0,1,2,\ldots)$.

By Kleene theorem:

ipr.(0) = create-imp-proc.(fpd-v, fpd-r, prd, (tye, pre[ide/[ ]]))
ipr.(n+1) = create-imp-proc.(fpd-v, fpd-r, prd, (tye, pre[ide/ipr.(n)]))
ipr = U {ipr(n) | n = 0,1,2,…}

A

Recursion – how it works?

An example

```
proc power (val n,m as nn_integer ref p as nn_integer),
     if m=0 then p:=1 else m:=m-1; call power(val n,m ref p); p:=p*m fi
endproc
```

power.0.(n, m, p) = **if** m=0 **then** p:=1 **else** m:=m-1 ● abort ● p:=p*m **fi**
= **if** m=0 **then** p:=1 **else** abort **fi**
= m=0 ➔ $p=1=n^0$
m>0 ➔ ?

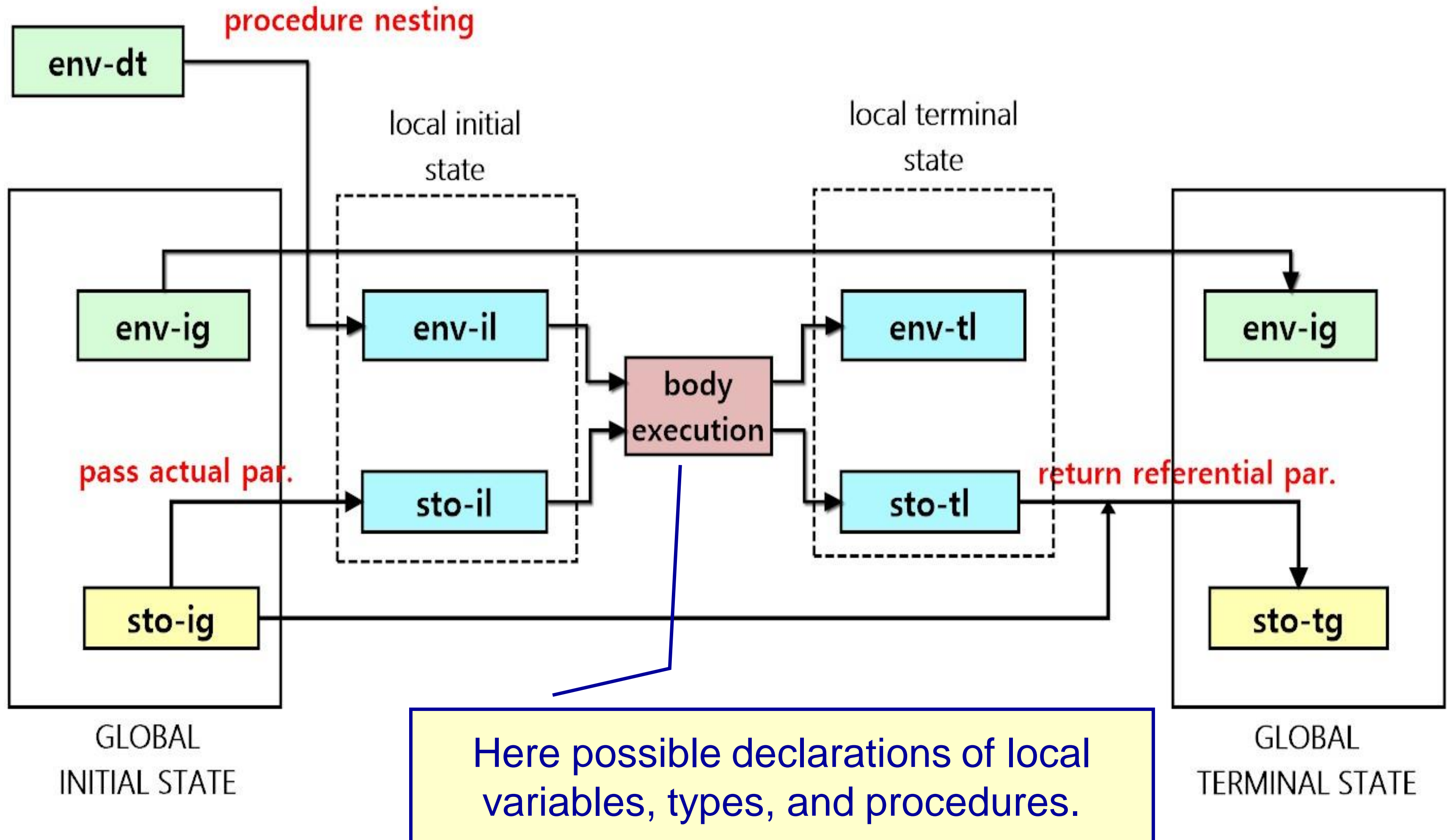
power.1 = **if** m=0 **then** p:=1 **else** m:=m-1 ● **ipr.0** ● p:=p*m **fi**
= **if** m=0 **then** p:=1 **else** m:=m-1 ●
if m=0 **then** p:=1 **else** abort **fi** ● p:=p*m **fi**
= m=0 ➔ $p=1=n^0$
m=1 ➔ $p=n=n^1$
m>1 ➔ ?

power.2.(n, m, p) = m=0 ➔ $p=1=n^0$
m=1 ➔ $p=n=n^1$
m=2 ➔ $p=n^2$
m>2 ➔ ?

A

The execution of a procedure call

reminder



Instruction of a procedure call

call-imp-proc : Identifier x AcPaDe x AcPaDe ⟼ InsDen   i.e.

call-imp-proc : Identifier x AcPaDe x AcPaDe ⟼ State → State

call-imp-proc.(ide, apd-v, apd-r).sta-ig =   - initial global state (-ig)

is-error.sta-ig ➔ sta-ig
let
(env-ig, sto-ig) = sta-ig
(tye-ig, pre-ig) = env-ig
pre-ig.ide = ? ➔ sta-ig ◄ 'procedure-unknown'
pre-ig.ide : FunPro ➔ sta-ig ◄ 'procedure-not-imperative'
let
ipr = pre-ig.ide   - the called imperative procedure
ipr.(apd-v, apd-r).sto-ig = ? ➔ ?   - infinite computation
let
sto-tg = ipr.(apd-v, apd-r).sto-ig   - terminal global store
is-error.sto-tg ➔ sta-ig ◄ error.sto-tg   - ig valuation unchanged
true ➔ (env-ig, sto-tg)

Imperative procedures with mutual recursion

P calls Q　　i.e.　　P = F(P,Q)
Q calls P　　　　　　Q = G(P,Q)

multiprocedure-component denotation

mcd : MprComDen = (Identifier x IprConDen)$^{c+}$

icd : IprConDen = FoPaDe x FoPaDe x ProDen (reminder)

imp.-procedure-content denotation

Two constructors:

create-mcd : Identifier x IprConDen $\mapsto$ MprComDen
create-mcd.(ide, icd) = ((ide, icd))　　- a one-element tuple of pairs

join-mcd : MprConDen x MprConDen $\mapsto$ MprConDen

join- mcd.(mcd-1, mcd-2) = mcd-1 © mcd-2

The constructor of procedure-calls the same as before.

Declaration of procedures with mutual recursion

declare-imp-mul-pro : MprComDen ↦ State ↦ State

declare-imp-mul-pro.mcd.sta =

is-error.sta ➔ sta

let

((ide-1, icd-1),…,(ide-n, icd-n)) = mcd

(fpd-v-i, fpd-r-i, prd-i) = icd-i for i=1;n

(env, (vat, err)) = sta

(tye, pre) = env

are-repetitions.(ide-1,…,ide-n) ➔ sta ◄ ‘procedure-names-are-repeated’

ide-i : declared.sta ➔ sta ◄ ‘identifier-ide-i-declared’ for i=1;n

let

ipr-1 = create-imp-proc.(icd-1, (tye, pre[ide-1/ipr-1,…,ide-n/ipr-n])

…

ipr-n = create-imp-proc.(icd-n, (tye, pre[ide-1/ipr-1,…,ide-n/ipr-n])

true ➔ ((tye, pre[ide-1/ipr-1,…,ide-n/ipr-n]), (sto, ‘OK’))

For n = 1 this definition coincides with single recursion.

multirecursion

Mutual recursion is "executed in parallel"

ipr-1.1 = create-imp-proc.(icd-1, (tye, pre[ide-1/[ ],…,ide-n/[ ]])

…

ipr-n.1 = create-imp-proc.(icd-n, (tye, pre[ide-1/[ ],…,ide-n/[ ]])

for any k = 0,1.,,,

ipr-1.(k+1) = create-imp-proc.(icd-1, (tye, pre[ide-1/ipr-1.k,…,ide-n/ipr-n.k])

…

ipr-n.(k+1) = create-imp-proc.(icd-n, (tye, pre[ide-1/ipr-1.k,…,ide-n/ipr-n.k])

Functional procedures

An example of a functional-procedure declaration

```
fun absolute-power(n, m integer)
 let p be integer ;
 p := 1;
 while m > 0 do p := p*n ; m:=m-1 od
 return if p ≤ 0 then -p else p fi as integer
endfun
```

program

type expression

data expression

Basic assumptions:

1. structure – a program (possibly **skip-d;skip-i**) followed by a data expression followed by a type expression,
2. no side-effects – no global variables or reference parameters,

Functional-procedures

fpr : FunPro = AcPaDe $\mapsto$ Store $\to$ ValueE - functional procedures

Functional-procedure calls belong to DatExpDen

Functional procedure declarations belong to DecDen

Functional-procedure content denotations

fcd : FprConDen = FoPaDe x ProDen x DatExpDen x TypExpDen

only value parameters

Declaration and instruction may be trivial

Exportation function (error considerations omitted for simplicity)

export : DatExpDen x TypExpDen $\mapsto$ State $\mapsto$ ValueE

export.(ded, ted).sta =
 let
  (dat-d, bod-d, yok-d) = ded.sta
  (bod-t, yok-t) = ted.sta
  com = (dat-d, bod-d)
 bod-t ≠ bod-d ➔ ‘bodies-inconsistent’
 yok-t.com ≠ (tt, (‘boolean’)) ➔ ‘yoke-not-satisfied’
 true ➔ ded.sta

either yok.com = (ff, 'Boolean')
or yok.com : Error

Constructor of functional procedures

create-fun-pro : FprConDen x Env ⟼ FunPro   i.e.

create-fun-pro : FoPaDe x ProDen x DatExpDen x TypExpDen) x Env

⟼ (AcPaDe ⟼ Store → ValueE)

create-fun-pro.((fop-v, prd, ded, ted), env-dt).apd-v.sto-ig =

is-error.sto-ig ➔ error.sto-ig   initial global store

let

(vat-ig, ‘OK’)) = sto-ig

(tye-dt, pre) = env-dt   declaration time env.

vat-il = pass-actual.(fop-v, (), apd-v, ()).(tye-dt, vat-ig)   initial local valuation

val-il : Error ➔ vat-il

let

sta-il = (env-dt, (vat-il, ‘OK’))

prd.sta-il = ? ➔ ?

let

sta-tl = prd.sta-il   terminal local store

is-error.sta-tl ➔ error.sta-tl

true ➔ export.(ded, ted).sta-tl

Calls of functional procedures (informally)

1. getting the called procedure from an environment,
2. computing the values of its actual parameters,
3. applying the procedure to parameters in order to get an expression denotation; this denotation is responsible for passing the values of actual parameters to formal parameters,
4. applying this denotation to the actual state which — if the computation terminates — returns a value or an error message.

Calls of functional procedures

call-fun-pro : Identifier x AcPaDe $\mapsto$ DatExpDen or:
call-fun-pro : Identifier x AcPaDe $\mapsto$ State $\to$ ValueE

call-fun-pro.(ide, apd).sta =

```
  is-error.sta          ➔ error.sta
  let
    ((tye, pre), sto)  = sta
  pre.ide = ?           ➔ 'procedure-not-declared'
  pre.ide : ImpPro      ➔ 'procedure-not-functional'
  let
    fpr = pre.ide                                  - functional procedure
  fpr.apd.sto = ?       ➔ ?
  true                  ➔ fpr.apd.sto
```

Declarations of functional procedures

declare-fun-pro : Identifier x FprConDen ⟼ State ⟼ State i.e.

declare-fun-pro : Identifier x FoPaDe x ProDen x DatExpDen x TypExpDen ⟼
State ⟼ State

declare-fun-pro.(ide, (fpd, prd, ded, ted)).sta =

is-error.sta ➔ error.sta

ide : declared.sta ➔ sta ◄ ‘variable-declared’

let

((tye, pre), sto) = sta

fpr = create-fun-pro.(fpd, prd, ded, ted, (tye-dt, pre-dt[ide/fpr]) fixed-point recursion

true ➔ ((tye, pre[ide/fpr]), sto)

Procedures as parameters of procedures

An intuitive (illegal) solution

Procedure = Parameter ⟼ Store → Store

Parameter = Composite | Procedure

A hierarchy of procedures – no self-application

Procedure.0 = Parameter.0 ⟼ Store → Store

Parameter.0 = AcPaDe x AcPaDe

For $n \geq 0$:

Parameter.(n+1) = Parameter.0 | … | Parameter.n

Procedure.n = Parameter.n ⟼ Store → Store

Concrete syntax of Lingua-2
(a recapitulation of denotational carriers)

ide : Identifier		- identifiers
ded : DatExpDen		- data-expression denotations including new: function calls
tra : TraExpDen		- transfer-expression denotations
yok : YokExpDen		- yoke-expression denotations
ted : TypExpDen		- type-expression denotations
din : InsDen		- instruction denotations including new: procedure calls
fpd : FoPaDe	new	- formal parameter denotations
apd : AcPaDe	new	- actual parameter denotations
icd : IprConDen	new	- imperative-procedure content denotations
mcd: MprComDen	new	- multiprocedure-component denotations
fcd : FprConDen	new	- functional-procedure content denotations
dde : DecDen		- declaration denotations new: procedure declarations
prd : ProDen		- program denotations new declarations and instructions

Concrete syntax of Lingua-2

the grammar

ide : Identifier = (as in Lingua-A)
tex : TypExp = (as in Lingua-A)
dae : DatExp = (as in Lingua-A) | Identifier (ActPar) - functional-proc. call

ins : Instruction = (as in Lingua-1) | **call (** Identifier (**val** ActPar **ref** ActPar) **)**

acp : ActPar = empty-ap | Identifier | **(** ActPar , Identifier **)**
fop : ForPar = empty-fp | Identifier **as** TypExp | ForPar , Identifier **as** TypExp

ico : IprCon = **((val** ForPar **ref** ForPar**)** Program **)**
mpc: MprCon = (Identifier , IprCon) | (MprCon **,** MprCon)
fco : FprCon = Identifier (ForPar) Pro **return** (DatExp **as** TypExp)

dec : Declaration=
(variable declarations and type declarations as in **Lingua-1**) |
proc Identifier , IprCon **endproc** |
mulproc MprCon **endmulproc** |
fun Identifier , FprCon **endfun**

prg : Program = Declaration **;** Instruction (as in **Lingua-1**)

A comment about lists of parameters in colloquial syntax

acp : ActPar = `empty-ap` | Identifier | ActPar , Identifier

Consequences:

(1) A list of actual parameters may be of the form

```
empty-ap, x, y, z
```

(2) We can't write

```
call inventory (ref x, y)
```

but we have to write

```
call inventory (val empty-ap ref x, y)
```

In Lingua every information should be explicite.

Colloquial syntax of Lingua-2

(1) All colloquialisms of Lingua-1

(2) Abbreviations for **as**

```
proc name (val x, y, z as integer ref a, b, c as word)
```

Thank you for
your attention