

MICHAEL LEUSCHEL

# PARTIAL EVALUATION: A TUTORIAL

02/02/2012

# OVERVIEW

- Partial Evaluation versus Full Evaluation
- Why PE
- Issues in PE: Correctness, Precision, Termination
- Self-Application and Compilation

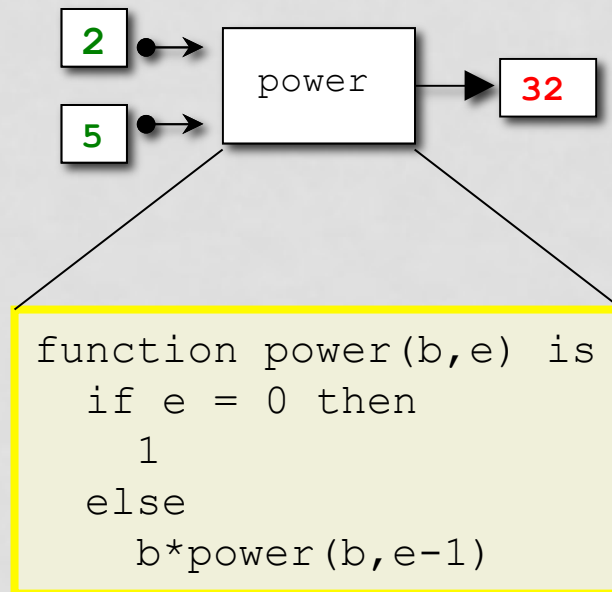


PART 1:

PARTIAL EVALUATION VS. FULL  
EVALUATION

# FULL EVALUATION

- Full input
- Computes full output



# PARTIAL EVALUATION

- Only part of the input
- → ~~produce part of the output~~ ?

`power (? , 2)`

- → evaluate as much as you can
- → produce a specialized program

`power_2 (?)`

# EXAMPLE: POWER

- **power**(?, 2)

```
function power(b,2) is
  if false then
    1
  else
    b*power(b,2-1)
```

Residual code:

```
function power_2(b) is
  b*power_1(b)
```

- **power**(?, 1)

```
function power(b,1) is
  if false then
    1
  else
    b*power(b,1-1)
```

```
function power_1(b) is
  b*power_0(b)
```

# EXAMPLE: POWER (CONT'D)

- `power(?, 0)`

```
function power(b, 0) is
  if 0 = 0 then
    1
  else
    b*power(b, e-1)
```

Residual code:

```
function power_0(b) is
  1
```

# EXAMPLE: POWER (CONT'D)

- Residual code:

```
function power_2(b) is  
  b*power_1(b)
```

```
function power_1(b) is  
  b*power_0(b)
```

```
function power_0(b) is  
  1
```

- What we really want:

```
function power_2(b) is  
  b*b
```

# UNFOLDING

- Replace call by definition

```
function power(b,2) is
  if 2 = 0 then
    1
  else
    b* (if 1 = 0 then 1
        else b* (if 0 = 0 then 1
                 else b*power(b,-1)))
```

Similar to  
loop-unrolling

Residual code:

```
function power_2(b) is
  b*b*1
```

PART 2:

WHY PARTIAL EVALUATION ?

# CONSTANT FOLDING

- **Static** values in programs

```
function my_program(x) is
...
let y = 2* power(x, 3)
...
```

captures inlining and more

```
function my_program(x) is
...
let y = 2* x * x * x
...
```

# ABSTRACT DATATYPES / MODULES

- Get rid of inherent overhead by PE

```
function my_program(x) is
...
if not empty_tree(x) then
  let y = get_value(x)...
```

+ get rid of redundant run-time tests

```
function my_program(x) is
...
if x \= null then
  let x=tree(_,y,_) ...
```

# HIGHER-ORDER/REUSING GENERIC CODE

- Get rid of overhead → incite usage

```
function my_program(x, y, z) is
...
let r = reduce(* , 1, map(inc, [x, y, z]))
...
```



```
function my_program(x, y, z) is
...
let r = (x+1) * (y+1) * (z+1)
...
```

# HIGHER-ORDER II

- Can generate new functions/procedures

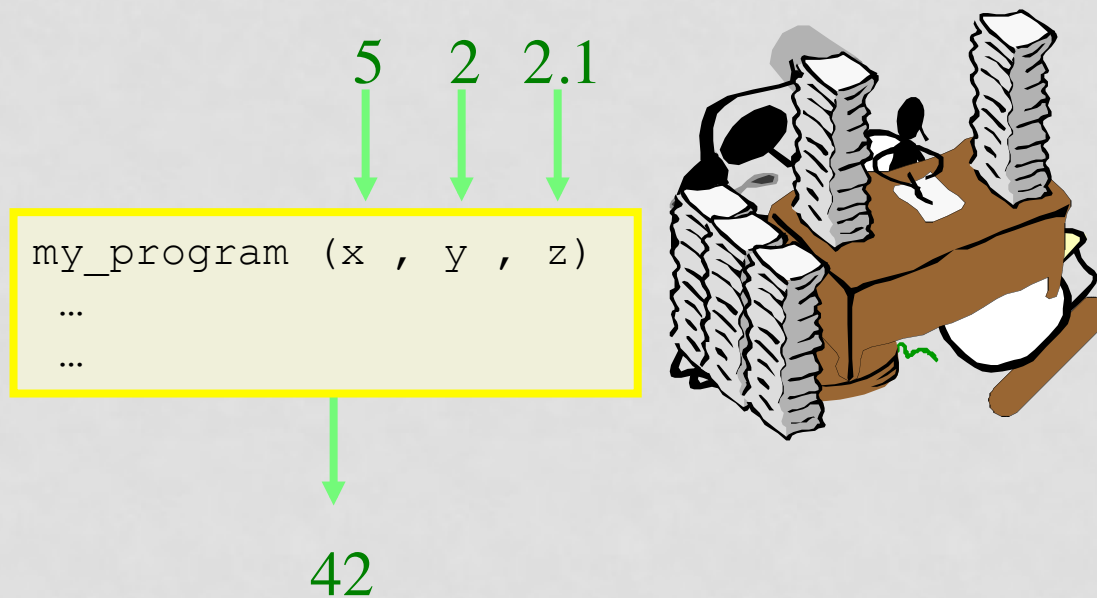
```
function my_program(x) is
...
let r = reduce(* , 1, map(inc, x))
...
```



```
function my_program(x) is
...
let r = red_map_1(x)
...
function red_map_1(x) is
if x=nil then return 1
else return head(x) * red_map_1(tail(x))
```

# STAGED INPUT

- Input does not arrive all at once



# EXAMPLES OF STAGED INPUT

- Ray tracing

```
calculate_view(Scene, Lights, Viewpoint)
```

## Interpretation

```
interpreter(ObjectProgram, Call)
```

```
prove_theorem(FOL-Theory, Theorem)
```

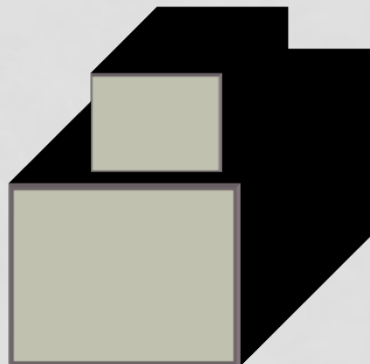
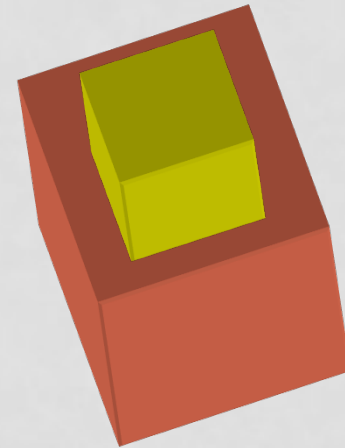
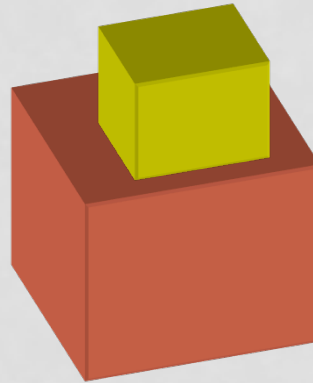
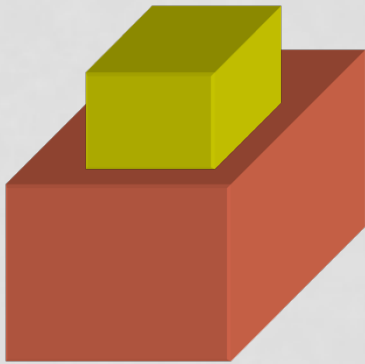
```
check_integrity(Db_rules, Update)
```

```
schedule_crews(Rules, Facts)
```

- Speedups

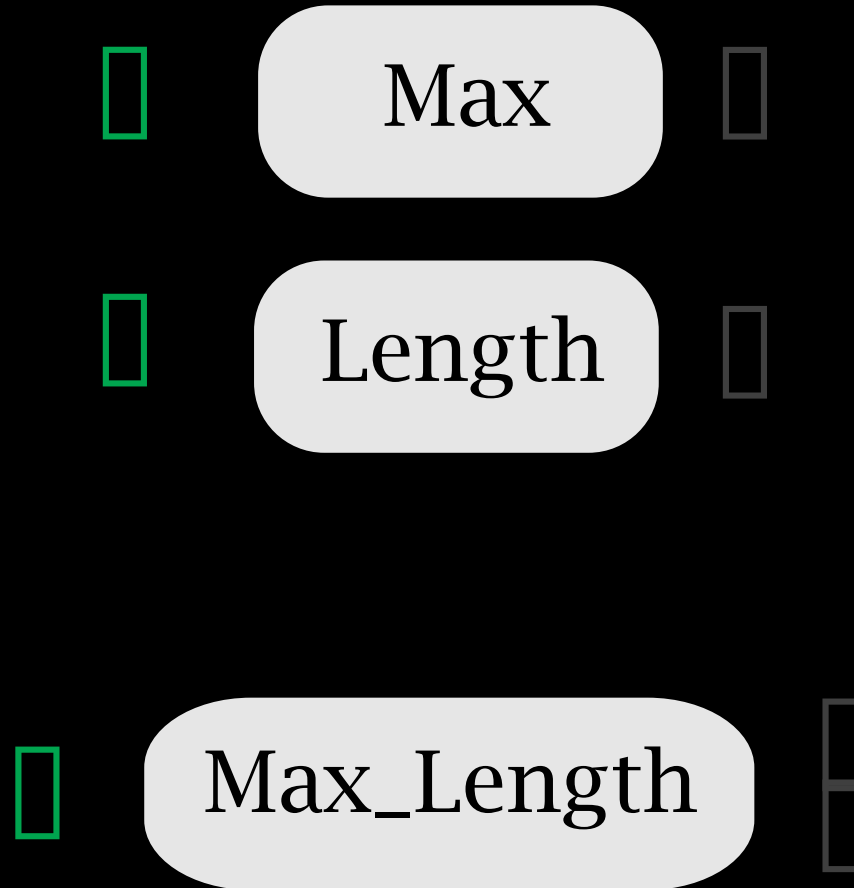
- 10: quite typical for interpretation overhead
- 100-500 (and even  $\infty$ ): possible

# RAY TRACING



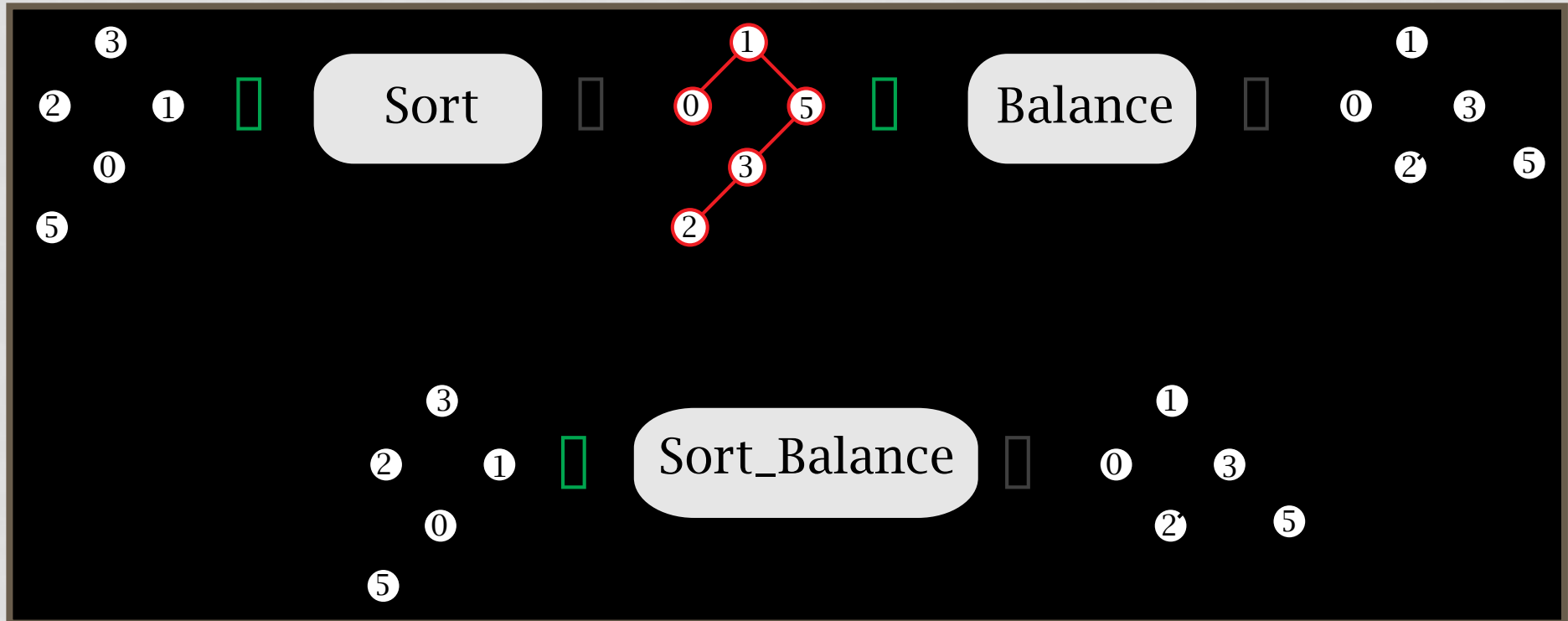
Static

# TUPLING



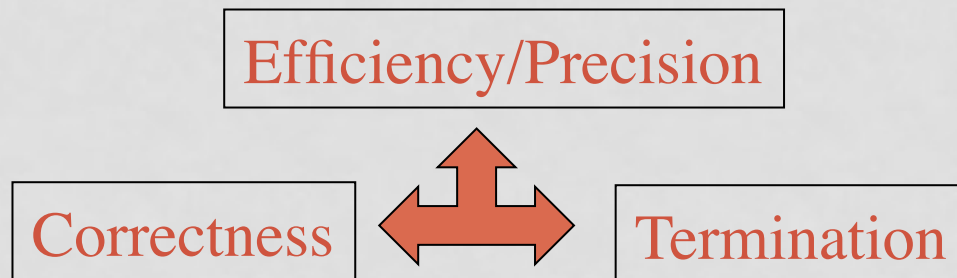
Corresponds to  
loop-fusion !?

# DEFORESTATION

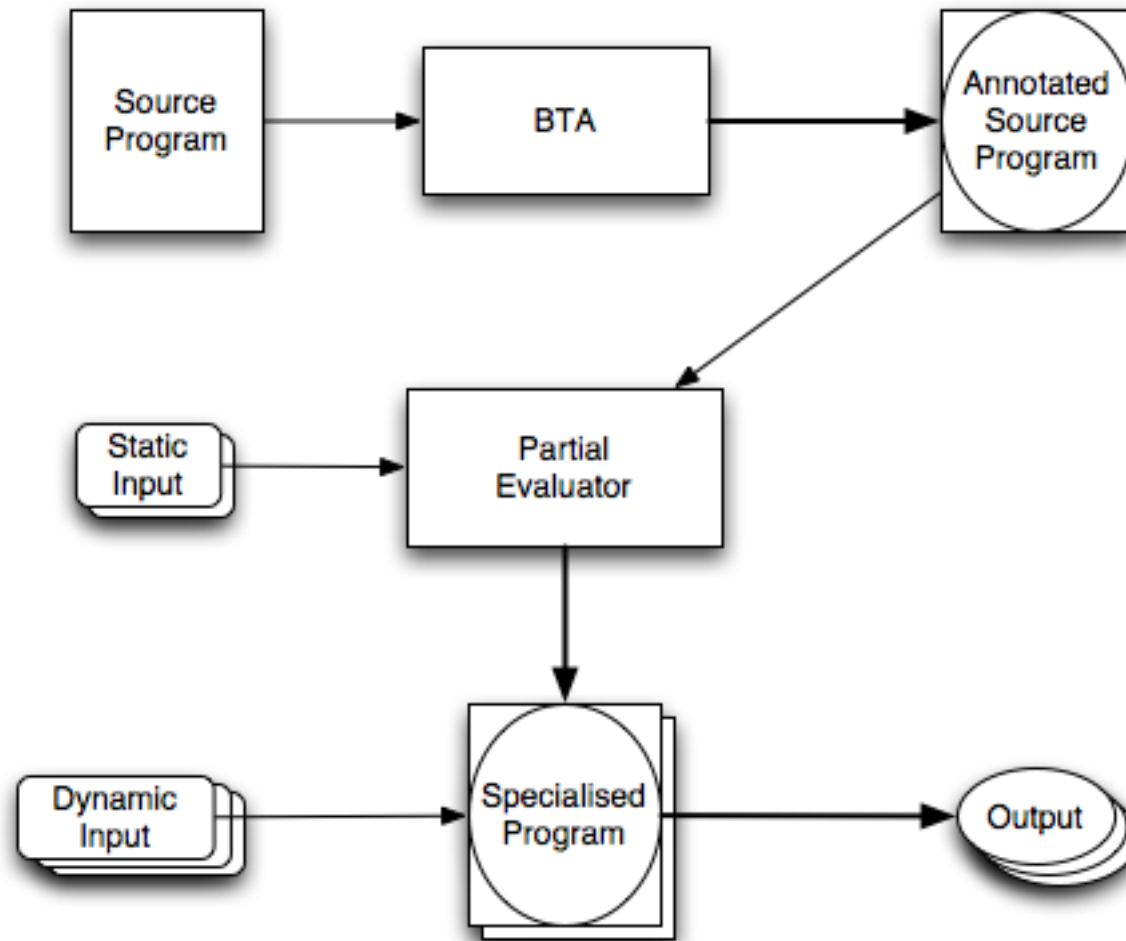


## PART 3:

# ISSUES IN PARTIAL EVALUATION



# OFFLINE VS ONLINE SPEZIALISATION



# CORRECTNESS

- Language
  - Power/Elegance: unfolding LP easy, FP ok, C++ aargh
  - Modularity: global variables, pointers
- Semantics
  - Purely logical
  - Somewhat operational (termination,...)
  - Purely operational
  - Informal/compiler

admissive





restrictive

# EFFICIENCY/PRECISION

- Unfold enough but not too much
  - Binding-time Analysis (for offline systems):
    - Which expressions will be definitely known
    - Which statements can be executed
  - Termination
- Allow enough polyvariance but not too much
  - Characteristic trees

# TERMINATION

- Who cares
- PE should terminate when the program does
- PE should always terminate  State of the art
- " within reasonable time bounds

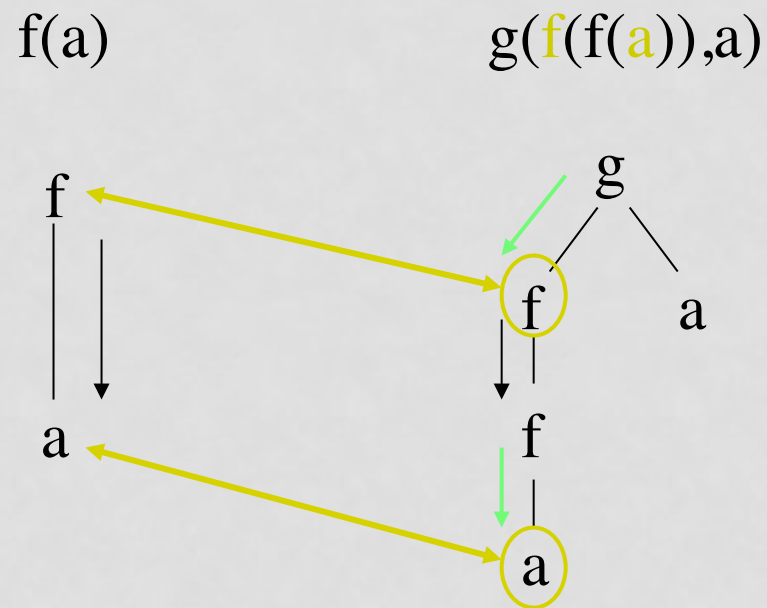
 Well-founded orders  
Well-quasi orders

# WFO'S AND WQO'S

- No  $\infty$  descending chains  $s_1 < s_2 < s_3 < \dots$
- Define  $<$  on expressions
- Ensure that  $s_{k+1} < s_k$
- termsize, linear norms, lexicographic ordering, recursive path ordering, ...
- More efficient
- Can be used statically
- Every  $\infty$  sequence:  $s_i \leq s_j$  with  $i < j$
- Define  $\leq$  on expressions
- Ensure that not  $s_i \leq s_{k+1}$
- homeomorphic embedding  $\diamond$
- More powerful (SAS'98)
- $\diamond$  more powerful than all monotonic and simplification orderings

# HOMEOMORPHIC EMBEDDING ❖

- **Diving:**  $s \trianglelefteq f(t_1, \dots, t_n)$  if  $\exists i: s \trianglelefteq t_i$
- **Coupling:**  $f(s_1, \dots, s_n) \trianglelefteq f(t_1, \dots, t_n)$  if  $\forall i: s_i \trianglelefteq t_i$   $n \geq 0$



# HIGMAN-KRUSKAL THEOREM (1952/60)

- $\preceq$  is a WQO (over a finite alphabet)
- Infinite alphabets + associative operators:  
 $f(s_1, \dots, s_n) \preceq g(t_1, \dots, t_m)$  if  $f \leq g$  and  
 $\forall i: s_i \preceq t_{j_i}$  with  $1 \leq j_1 < j_2 < \dots < j_n \leq m$

$$\text{and}(q, p(b)) \preceq \text{and}(r, q, p(f(b)), s)$$


- Variables :  
 $X \preceq Y$   
more refined solutions possible (DSSE-TR-98-11)

# AN EXAMPLE FOR $\leq$

- `rev([a,b | X], [])`
- `rev([b | X], [b])`
- `rev(X, [b,a])`
- `rev(Y, [H,b,a])`



- `eval(rev([a,b | X], []), 0)`
- `eval(rev([b | X], [b]), s(0))`
- `eval(rev(X, [b,a]), s(s(0)))`
- `eval(rev(Y, [H,b,a]), s(s(s(0))))`



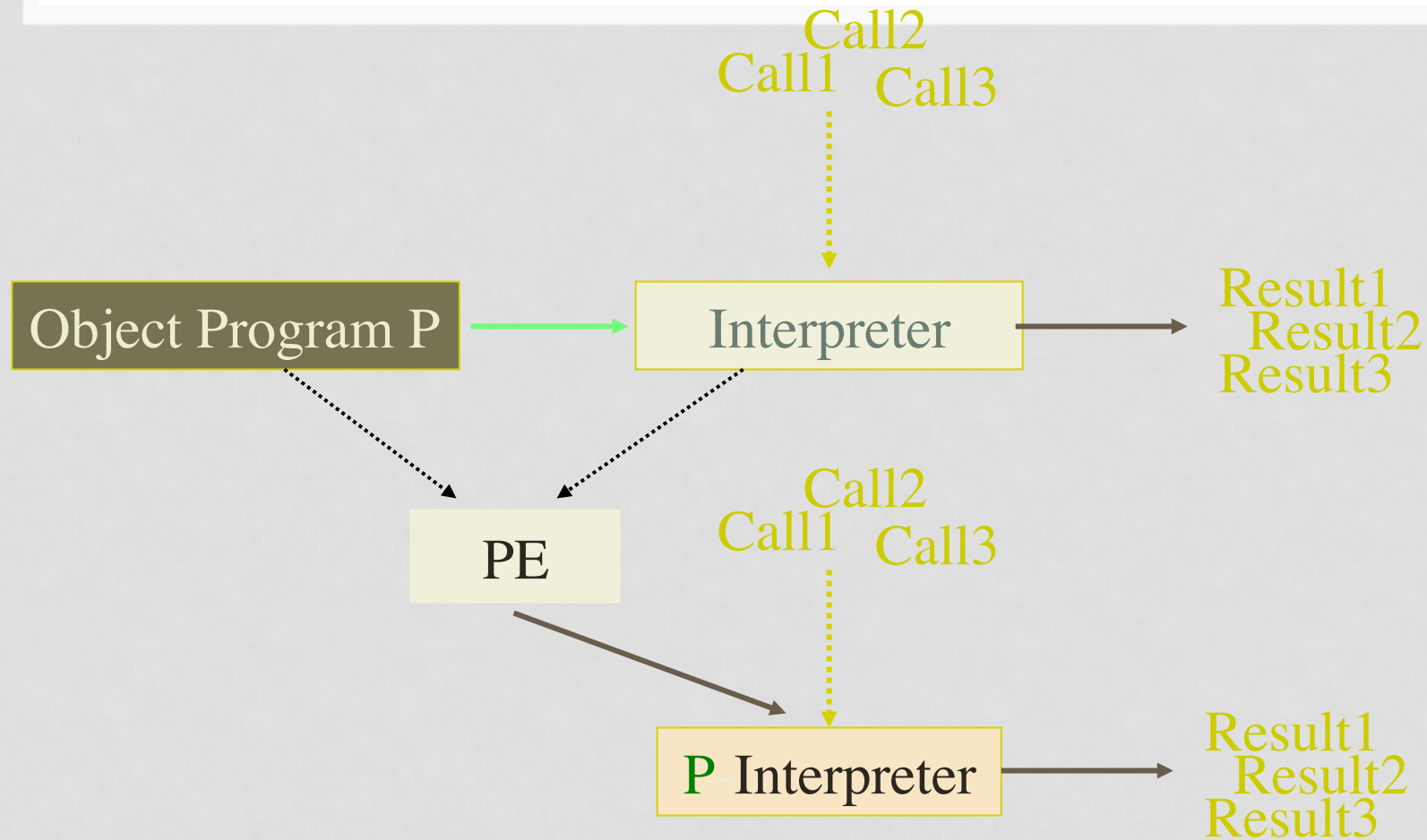
- ✓ Data consumption
- ✓ Termination
- ✓ Stops at the “right” time
- ✓ Deals with encodings

PART 4:

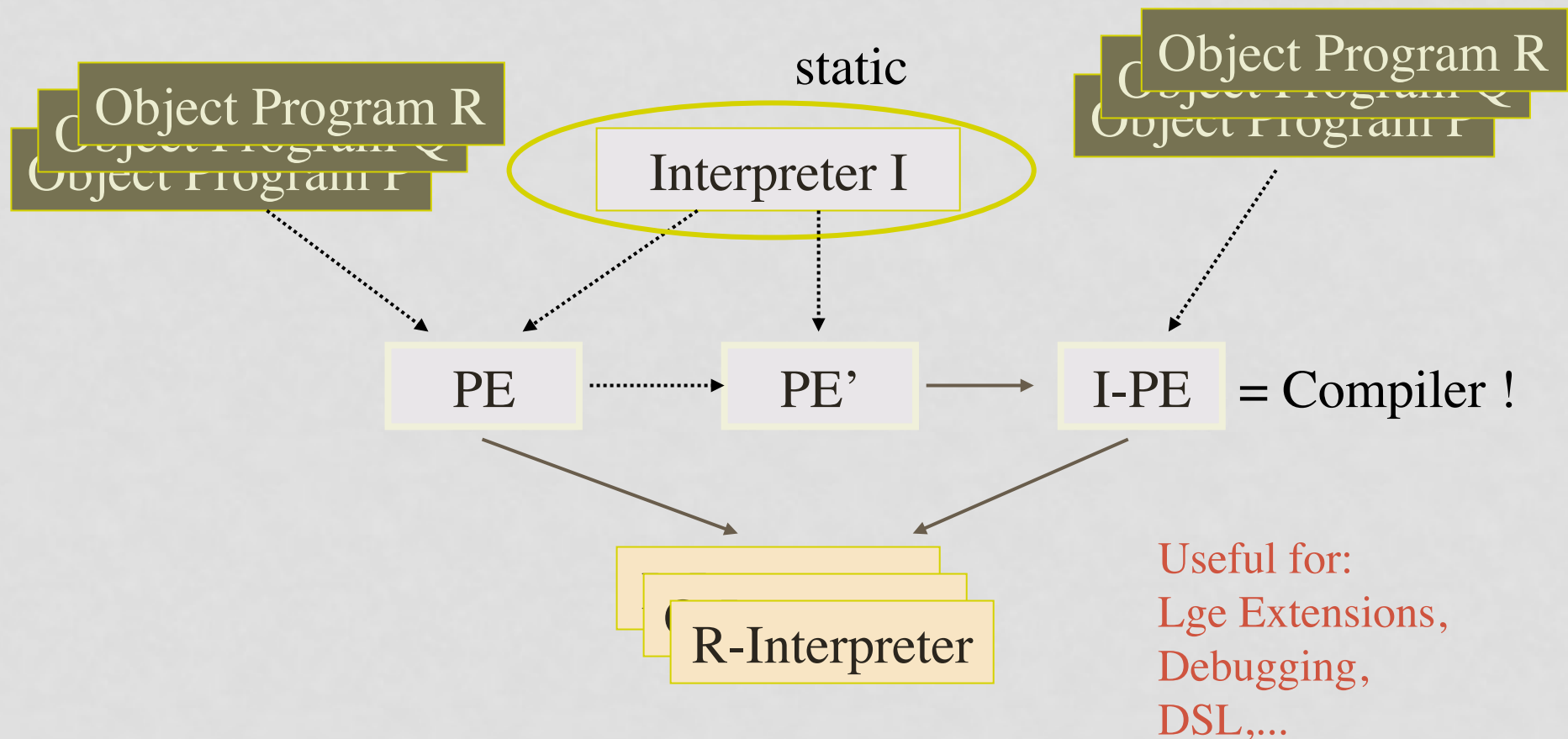
SELF-APPLICATION AND  
COMPILATION



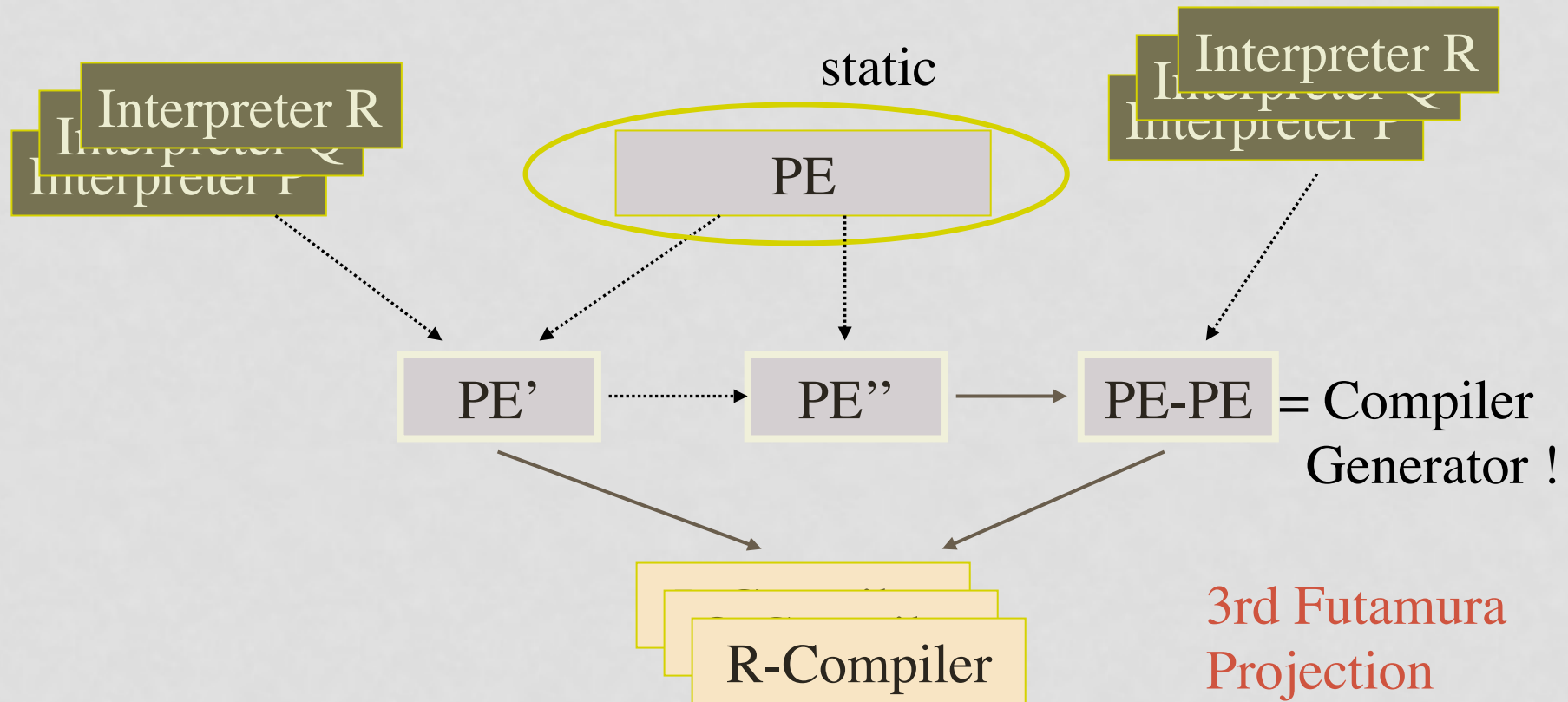
# COMPILING BY PE



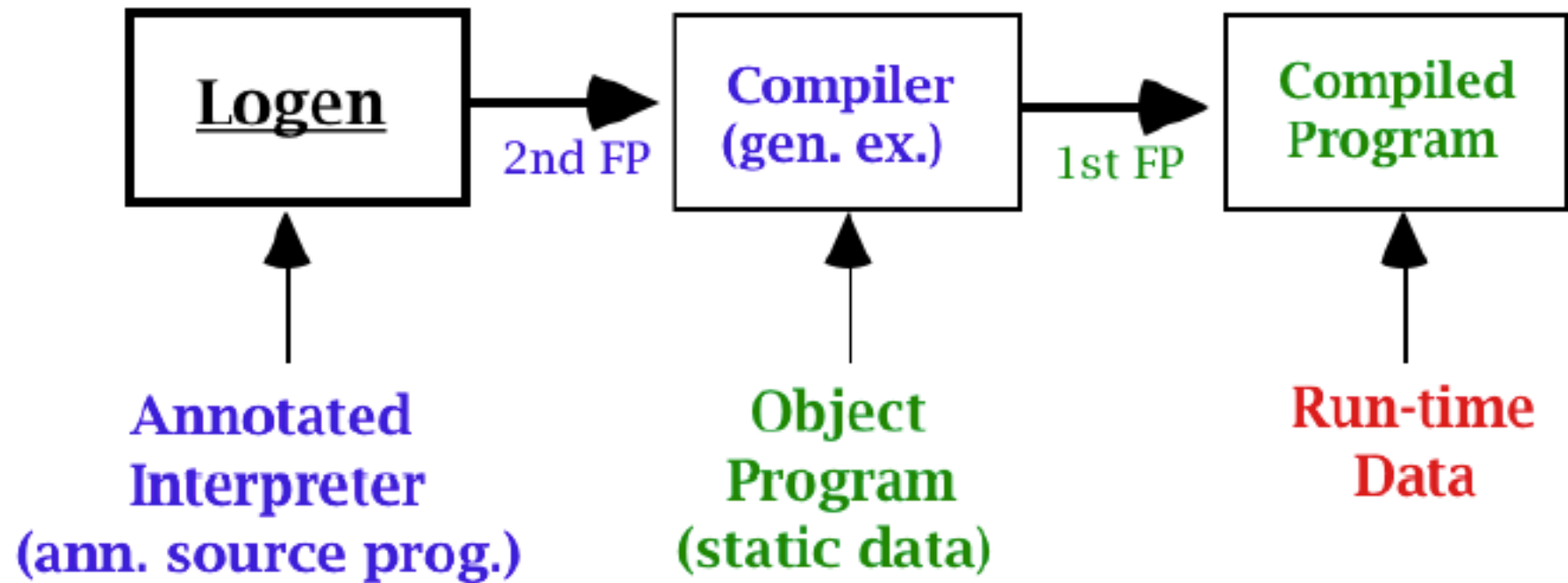
# COMPILER GENERATION BY PE



# COGEN GENERATION BY PE



# WHAT IS A COGEN

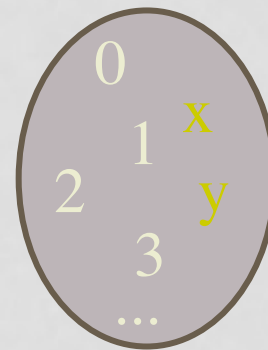


# COGEN BY HAND

- Writing a self-applicable PE is difficult
- Idea: write the Cogen instead of the PE:
  - No worry about self-application
  - Easier than one might think (BTA)
  - Works for FP (Romanenko, Holst&Launchbury), LP (Jørgensen&Leuschel), C (Andersen)
  - Very fast specialisation (runtime code-generation possible: 0.006 s to compile ground repr. of append)

# BINDING-TIME ANALYSIS

- Perform abstract interpretation over domain  $\{\text{static}, \text{dynamic}\}$



Concrete



Abstract  
domain

- static + static = static
- dynamic + static = dynamic
- ...

# SUMMARY

- Basics of PE
- Uses of PE
  - Common compiler optimisations
  - Enabling high-level programming
  - Staged input, optimising existing code
- Some issues in PE
  - termination:  $wqo \preceq$
- Self-application and compiler generation

# WHERE TO GET MORE INFORMATION

- Play with **ecce** and **logen**
- Tutorials (Consel&Danvy POPL'93, Neil Jones ACM Computing Surveys Sep'96, John Gallagher PEPM'93,...)
- Books
  - “Partial Evaluation and Automatic Program Generation” (Prentice-Hall,93)
  - “Partial Evaluation: Theory and Practice” (Springer, to appear)



OPTIONAL PART 5:

PE VERSUS PROGRAM  
SPECIALISATION

# PROGRAM SPECIALISATION

- Specialise a program for a particular context
- Ways to specialise a program:
  - Partial evaluation
  - Abstract Interpretation
  - Type inference
  - Slicing