

Multi-stage Programming in MetaOCaml

Walid Taha

Rice University

Why Multi-stage Programming?

Reasons are purely economic:

- Problem 1: Abstraction mechanisms (functions, objects, classes...) have runtime cost
 - Result 1: You don't use them
 - Result 2: That costs you programmer productivity
- Idea: Use generative programming (GPCE)
- Problem 2: Writing good generative programs is hard

Why program generation is hard

- First, there is the issue of hygiene (as in macros)
- More generally, what can we statically guarantee about what's generated?
- Important: “Statically” = by looking at the generator?

Approach	Build “f (x,y)”	Combine F X	Syntactic correctness? Reject “f (x,)”	Type correctness? Reject “7 (8)”
String	✓	✓	✗	✗
Datatype	✗	✓	✓	✗

Multi-stage programming (MSP)

- Provide abstraction mechanisms like: polymorphism, higher-order functions, exceptions, ...
- Provide constructs that allow “generation”
- Without damaging the static typing discipline

Approach	Build “f (x,y)”	Combine F X	Syntactic correctness? Reject “f (x,)”	Type correctness? Reject “7 (8)”
String	✓	✓	✗	✗
Datatype	...	✓	✓	✗
MSP	✓	✓	✓	✓

Multi-stage programming (MSP)

Three staging annotations:

Construct	Example	Result
Brackets:	$a = \langle 2 * 4 \rangle$	$a = \langle 2 * 4 \rangle$
Escape:	$b = \langle 9 + \sim a \rangle$	$b = \langle 9 + 2 * 4 \rangle$
Run:	$c = !b$	$c = 17$

Typing rules (simplified):

Term	Type	Term	Type
x	T	$\langle x \rangle$	$\langle T \rangle$
x	$\langle T \rangle$	$\sim x$	T
x	$\langle T \rangle$	$!x$	T

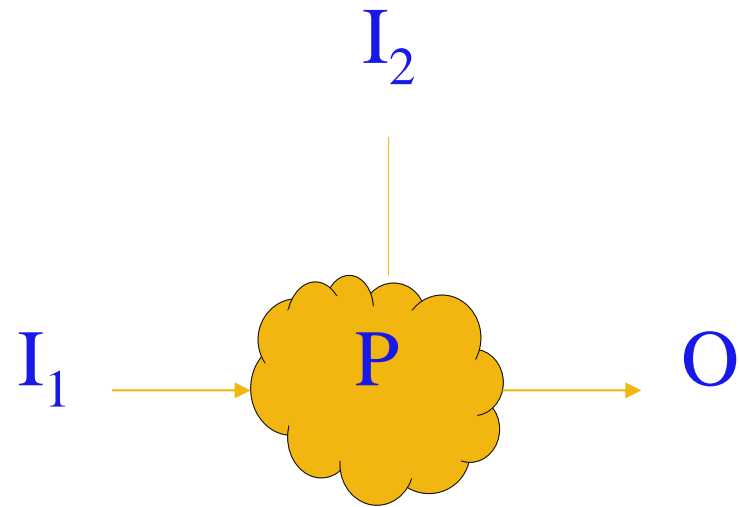
Related techniques for “staging”

There is a number of ways for looking at MSP:

- Statically typed macros
 - But not limited to compile-time
- Partial evaluation (Mix, Schism, Similix, Tempo)
 - But programmer has full control over what is specialized
 - (in some cases, programmer has too much control)
- High-level program generation (SDRR, Genvoca, P++)
- Runtime code generation (Fabius, DyC, Dynamo, `C)

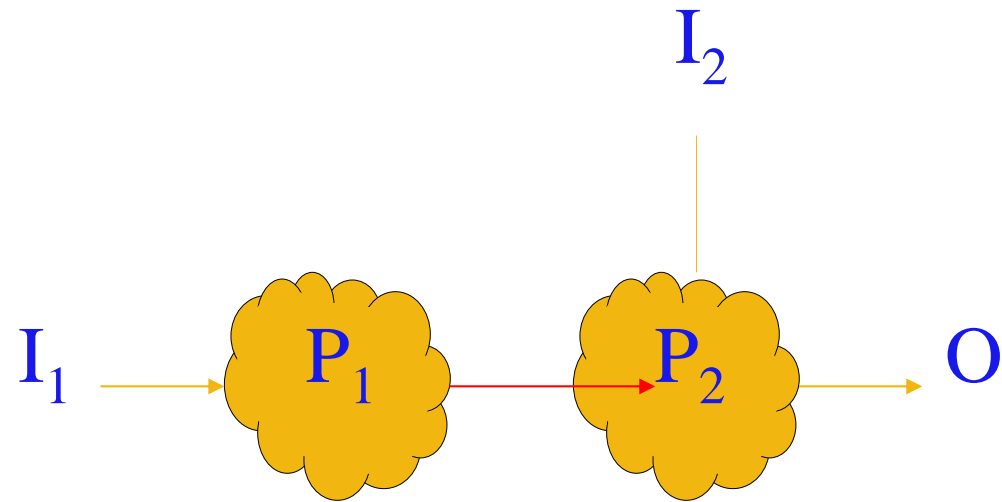
Common goal: ***Alter order of evaluation to reduce cost of a computation***

The abstract view



Batch Programming
Ex.: Interpreter

The abstract view



Multi-Stage Programming (MSP)
Ex.: Compiler

Where can staging be used?

Lots of applications, including:

- Operating systems [Pu et al]
- Data base systems [Batory et al]
- Satellite software [Czarnecki et al]
- Domain specific languages [Consel et al]
- Graphics software [Mogensen]

No shortage in applications. Rather, in methodology

Overview of this presentation

1. Staging the exponentiation function
 - A naïve model for writing MSP programs
 - Two steps: 1) write program, 2) stage it
 - Basic issues: Binding time, timing, termination, in-lining, code explosion
2. Staging a Markov chain problem
3. Staging a little interpreter (`lint`)
 - Binding time improvements. Adding AOP
4. Summary and “to probe further” pointers

Overview of this presentation

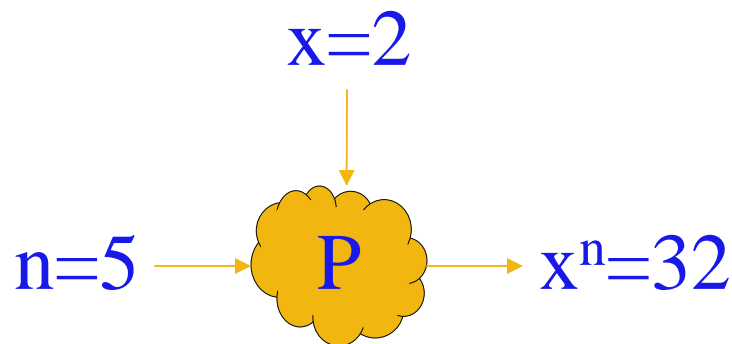
1. Staging the exponentiation function
 - A naïve model for writing MSP programs
 - Two steps: 1) write program, 2) stage it
 - Basic issues: Binding time, timing, termination, in-lining, code explosion
2. Staging a Markov chain problem
3. Staging a little interpreter (`lint`)
 - Binding time improvements. Adding AOP
4. Summary and “to probe further” pointers

Small example: Exponentiation

$$\begin{aligned}x^0 &= 1 \\x^{2n+1} &= x * x^{2n} \\x^{2n} &= (x^n)^2\end{aligned}$$

Minimal example of a generic program.

Such a program is almost never used in practice.



Why?

Small example: Exponentiation

$$x^0 = 1$$

$$x^{2n+1} = x * x^{2n}$$

$$x^{2n} = (x^n)^2$$

$$P_2(x) = \underline{x^5}$$

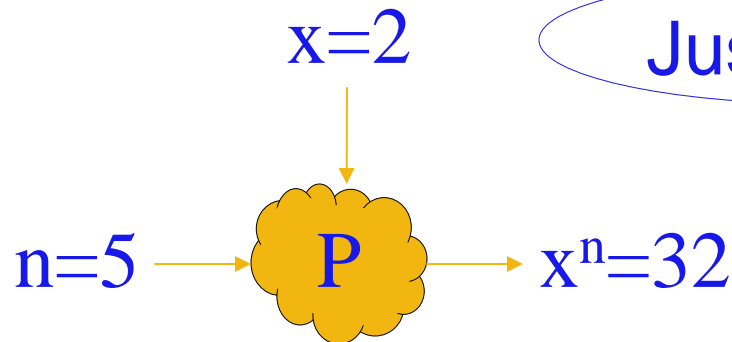
$$= x * \underline{x^4}$$

$$= x * (\underline{x^2})^2$$

$$= x * ((\underline{x^1})^2)^2$$

$$= x * ((x * \underline{x^0})^2)^2$$

$$= x * ((x * 1)^2)^2$$



Just 4 ops!

No recursion!

Small example: Exponentiation

$$x^0 = 1$$

$$x^{2n+1} = x * x^{2n}$$

$$x^{2n} = (x^n)^2$$

$$P_2(x) = \underline{x^5}$$

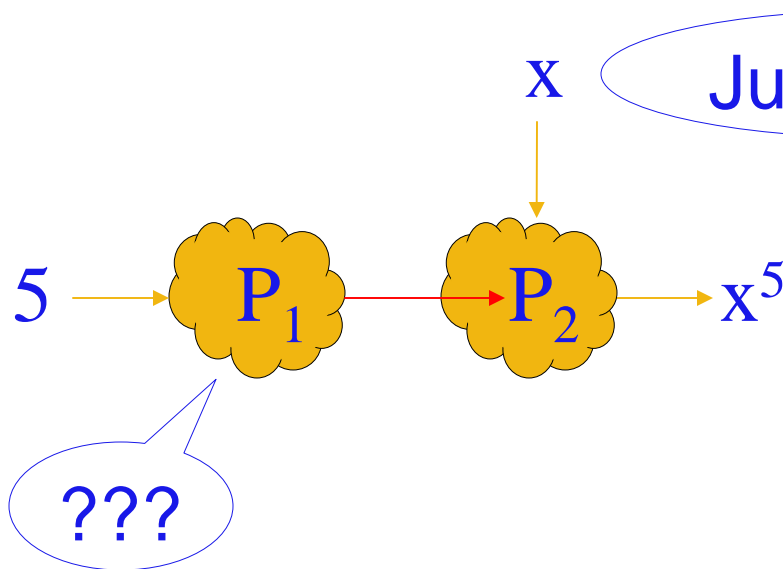
$$= x * \underline{x^4}$$

$$= x * (\underline{x^2})^2$$

$$= x * ((\underline{x^1})^2)^2$$

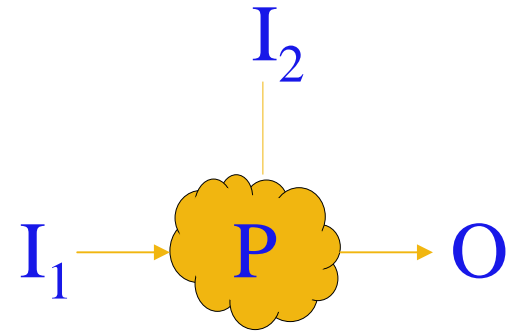
$$= x * ((x * \underline{x^0})^2)^2$$

$$= x * ((x * 1)^2)^2$$

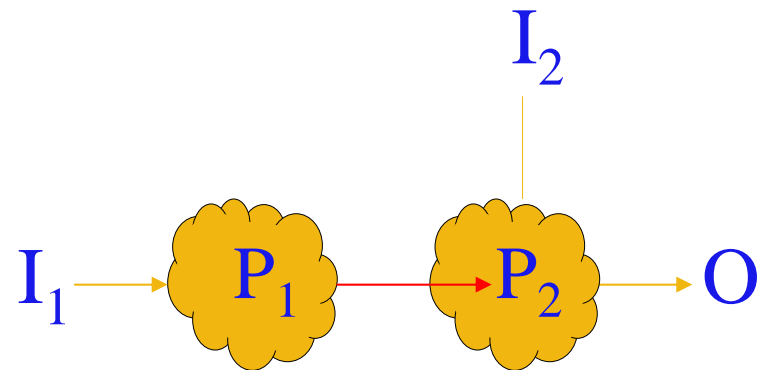


A Naïve MSP method

1. Write traditional, single-stage program



2. Add staging annotations:
brackets, escapes,
and run



MetaOCaml

[GPCE '03]

- Builds on the OCaml system
 - Very successful, widely used, statically typed language
 - Compiled (both byte-code and native code)
 - Has competitive performance (even for numerical apps)
 - Allows the collection of credible performance numbers
 - Substantial existing code base
- Realized by a source-to-source translation
 - Based on an elegant formal model
 - Has a clear underlying cost model (parse tree)

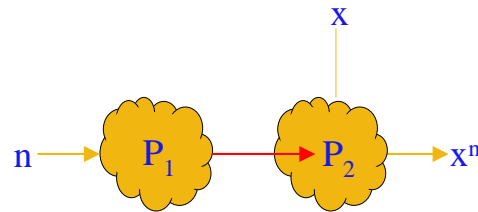
Small example: Exponentiation



```
let rec exp(n:int, x:real):real =  
if n = 0  
  then 1.0  
  else if even (n)  
    then sqr (exp(n div 2, x))  
    else x * (exp(n - 1, x));;
```

Small example: Exponentiation

Two stage:



```
let rec exp(n:int, x:<real>) : <real> =
```

```
if n = 0
```

```
then <1.0>
```

```
else if even (n)
```

```
then <sqr ~(exp(n div 2, x))>
```

```
else <~x * ~(exp(n - 1, x))>;;
```

Only things in
green left for
second stage

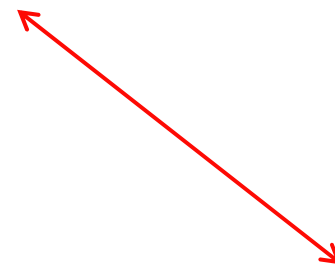
Small example: Exponentiation

To use the staged `exp` we simply write:

```
! <fn x => ~(exp(5, <x>)) > ; ;
```

The result is:

```
fn x => x*(sqr(sqr(x*1.0)))
```



$x*((x*1)^2)^2$

Timing in MetaOCaml (Careful!)

```
Trx.init_times ();; (* Initialize timing report *)  
let unstaged_result = Trx.timenew "Unstaged"  
    fun()-> p a b;;  
let code = Trx.timenew "Generating"  
    fun()-> <fun b->~(p1 a <b>)>;;  
let compilation = Trx.timenew "Compiling"  
    fun()-> .! code;;  
let staged_result = Trx.timenew "Stage 2"  
    fun()-> c2 b;;  
Trx.print_times ();; (* Prints timing report *)
```

Timing in MetaOCaml (Careful!)

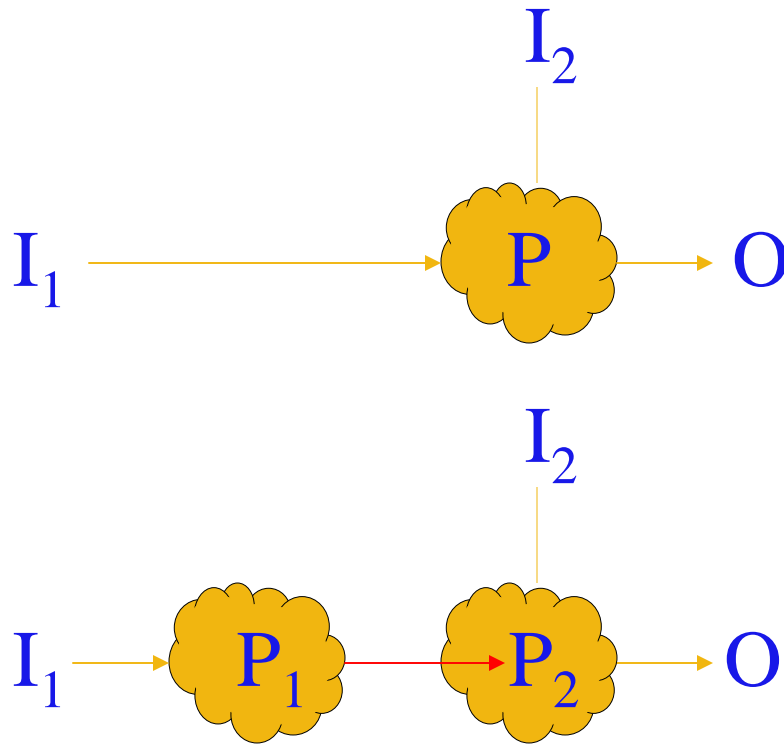
```
# #use "mex/3.ml";; (* loads "power" *)  
  
. . .  
__ Unstaged _____ 131072x avg= 2.400000E-03 msec  
__ Generating _____ 9192x avg= 5.670000E-02 msec  
__ Compiling _____ 512x avg= 2.000000E+00 msec  
__ Stage 2 _____ 8388608x avg= 3.000000E-04 msec  
- : unit = ()
```

The `timenew` function makes life easier

From the above data we can compute:

$$\text{"Speedup"} = (2.4\text{E-}3) / (3.0\text{E-}4) = 8 \times$$

Typical speedups (MetaOCaml)



Program	Speedup $T(P)/T(P_2)$
Exp	5...8 x
Interp(fib 15)	4 x
Interp'(fib 15)	44 x
RewriteCPS	13 x
Chebyshev	3 x

What kinds of programs can we stage?

Generic programs that pay an unnecessary cost for this generality.

“Delays lead to dangerous ends”

Be careful with recursion/loops... 

```
let rec exp(n:int, x:<real>) : <real> =
```

```
<if n = 0
```

```
  then 1.0
```

```
  else if even (n)
```

```
    then sqr ~ (exp(n div 2, x))
```

```
    else ~x * ~ (exp(n - 1, x))>;
```

Termination vs.
type safety

Basic inlining using MSP

Instead of getting

```
<fun x -> x * (sqr (sqr (x)))>
```

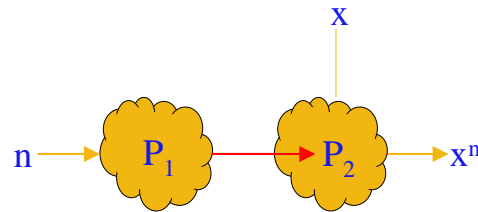
It would be nice if we can get

```
<fun x -> let y=x*x in x*(y*y)>
```

To do this, we need to consider the definition of `sqr`, (and change the staged `exp` function slightly)

Small example: Exponentiation

Two stage:



```
let rec exp(n:int, x:<real>) : <real> =
```

```
if n = 0
```

```
then <1.0>
```

```
else if even (n)
```

```
then <sqr ~(exp(n div 2, x))>
```

```
else <~x * ~(exp(n - 1, x))>;
```

Can we get rid
of this **sqr**
operation?

In-lining using MSP

Original definition

```
let sqr x = x * x
```

Naïve staged version

```
let sqr x = <~x * ~x>
```

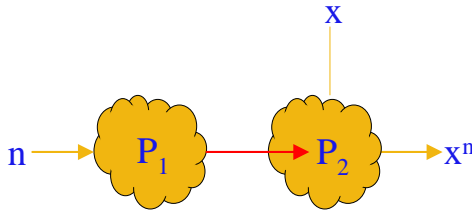
Note change in type:

Before: `int -> int` After: `<int> -> <int>`

Must therefore change `exp`

Small example: Exponentiation

Two stage:



```
let rec exp(n:int, x:<real>) :<real>
=
```

```
if n = 0
```

```
then <1.0>
```

```
else if even (n)
```

```
then sqr (exp(n div 2, x))
```

```
else <~x * ~(exp(n - 1, x))>;
```

Now our main
function has
even less green

Basic inlining using MSP

Problem: Using: `let sqr x = <~x * ~x>`

we get code explosion: We generate

```
<fun x -> x * ((x * x) * (x * x))>
```

Solution (from PE literature!)

```
let sqr x = <let y=~x in y*y>
```

Now we get (basically):

```
<fun x -> let y=x*x in x * (y*y)>
```

Partial Application vs. Partial Evaluation

“Curried application” is former, but NOT latter.

For example: $F\ x\ y\ z = x+y+z$

Is same as: $F\ x\ y = \text{fun } z \rightarrow x+y+z$

Now $F\ 1\ 2$ evaluates to $\text{fun } z \rightarrow 1+2+z$

but NOT: $\text{fun } z \rightarrow 3+z.$

Last step requires reduction under λ ($\text{fun } z$)

$F'\ x\ y = \langle \text{fun } z \rightarrow \sim(\text{lift}(x+y)) + z \rangle$

Capture is a non-problem in MSP

Given the following definitions

let $t(y) = \langle \text{fun } x \rightarrow x + \sim y \rangle$

let $p = \langle \text{fun } x \rightarrow \sim(t \langle x \rangle) 10 \rangle$

We do NOT get

$p = \langle \text{fun } x \rightarrow (\text{fun } x \rightarrow x+x) 10 \rangle$

But rather

$p = \langle \text{fun } x_1 \rightarrow (\text{fun } x_2 \rightarrow x_2+x_1) 10 \rangle$

With MSP, we don't need to worry about names

Why is type safety challenging?

Type system should reject “bad” terms:

- Basics: `~5, run 5`
- Levels: `<let f(x)=~x in ...>`
- Closedness: `<let f(x)=~(run<x>) in ...>`
→ `<let f(x)= ~x in ...>`

In the last case `x` is treated as having a code type, and `run` is used during “specialization”

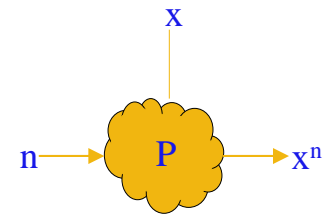
Summary of small example

- Writing generic programs with no runtime overhead
 - Basics of staging using MSP constructs
 - Gathering performance timing in MetaOCaml
 - How common pitfalls and of staging are avoided
 - variable capture and type safety (by language),
 - code explosion and termination (by programmer)
- How staging differs from partial application

Overview of this presentation

1. Staging the exponentiation function
 - A naïve model for writing MSP programs
 - Two steps: 1) write program, 2) stage it
 - Basic issues: Binding time, timing, termination, in-lining, code explosion
2. Staging a Markov chain problem
3. Staging a little interpreter (`lint`)
 - Binding time improvements. Adding AOP
4. Summary and “to probe further” pointers

A Markov chain problem



```
let mkmatrix rows cols p =
  let last_col = cols - 1
  and m = Array.make_matrix rows cols 0.0
  in
  for i = 0 to rows - 1 do
    let mi = m.(i) in
    for j = 0 to last_col do
      let r = rows / 4 in
      let mn = max 0 (i-r) in
      let mx = min rows (i+r) in
      let rn = mx-mn-1 in
      let q =(if i=j then p
              else if (j>=mn)&&(j<=mx) then
                (1.0 -. p)/.(float rn)
              else 0.0) in
      mi.(j) <- q;
    done;
  done;
  m
```

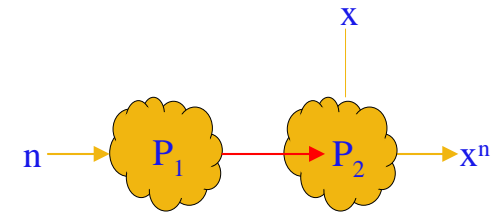
```
let rec inner_loop k v m1i m2 j =
  if k < 0 then v else inner_loop (k - 1)
    ((v+. (m1i.(k))) *. (m2.(k).(j))) m1i m2 j

let mmult rows cols m1 m2 m3 =
  let last_col = cols - 1
  and last_row = rows - 1 in
  for i = 0 to last_row do
    let m1i = m1.(i) and m3i = m3.(i) in
    for j = 0 to last_col do
      m3i.(j) <- inner_loop last_row 0.0 m1i
        m2 j done;
    done

let ans1 = fun p ->
  let size = 30 in
  let m1 = mkmatrix size size p and
      m2 = mkmatrix size size p in
  let m3 = Array.make_matrix size size 0.0
  in mmult size size m1 m2 m3; m3.(0).(0);;
```

A Markov chain problem

Staged model construction:



```
let lift x = <x>;      (* This is a non-trivial function *)
```

```
let mkmatrix rows cols p =
```

```
  let last_col = cols - 1
```

```
  and m = Array.make_matrix rows cols <0.0> in
```

```
  for i = 0 to rows - 1 do
```

```
    let mi = m.(i) in
```

```
    for j = 0 to last_col do
```

```
      let r = rows / 4 in
```

```
      let mn = max 0 (i-r) in
```

```
      let mx = min rows (i+r) in
```

```
      let rn = mx-mn-1 in
```

```
      let q =(if i=j then p
```

```
        else if (j>=mn)&&(j<=mx) then
```

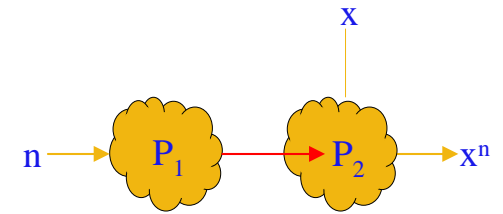
```
          <(1.0 -. ~p)/. (~(lift (float rn)))>
```

```
        else <0.0>) in
```

```
      mi.(j) <- q; done; done; m
```

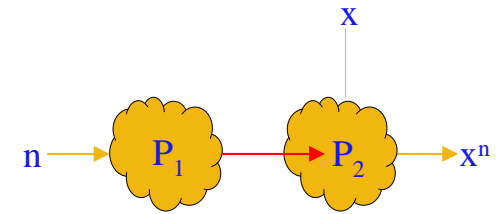
Code is a first-class value

A Markov chain problem



```
let mmult rows cols m1 m2 m3 =  
  let last_col = cols - 1 and last_row = rows - 1 in  
  for i = 0 to last_row do  
    let m1i = m1.(i) and m3i = m3.(i) in  
    for j = 0 to last_col do  
      m3i.(j) <- inner_loop last_row <0.0> m1i m2 j done;  
    done;;  
let ans = <fun p ->  
  ~(let size = 30 in  
    let m1 = mkmatrix size size <p> and  
      m2 = mkmatrix size size <p> in  
    let m3 = Array.make_matrix size size <0.0> in  
    mmult size size m1 m2 m3; m3.(0).(0))>;;
```

A Markov chain problem



Sample timings (for a 49-step solution)

```
___ r1 _____ 1x avg= 1.409207E+03 msec
___ ans _____ 2x avg= 6.094790E+02 msec
___ ans2 _____ 128x avg= 1.160370E+01 msec
___ r2 _____ 2048x avg= 6.519156E-01 msec
```

$$\text{Speedup} = 1.4\text{E}3 / 6.5\text{E}-1 > 2000x$$

A concern with imperative MSP

In the imperative MSP setting, we can have scope extrusion:

```
# let a = ref <1>;;  
# let b = <fun x -> ~(a := <x>; <x>)>;;  
# ! a;;  
- : int code = <x>
```

This is undesirable, and raise a runtime exception when this code is compiled (The exception can be handled)

Solutions published, but not implemented yet.

Summary of Markov example

- Speedups can be arbitrarily large
 - But remember, “speedup” is not the point
- Imperative programs, too, can be staged
 - Interesting type-system research questions
- Lift: a staging pattern, and when to use it
- MSP can be useful for numerical applications
 - Opportunity for experimental research

Overview of this presentation

1. Staging the exponentiation function
 - A naïve model for writing MSP programs
 - Two steps: 1) write program, 2) stage it
 - Basic issues: Binding time, timing, termination, in-lining, code explosion
2. Staging a Markov chain problem
3. Staging a little interpreter (`lint`)
 - Binding time improvements. Adding AOP
4. Summary and “to probe further” pointers

Domain Specific Languages (DSLs)

Examples

- Lex, yacc, VHDL, LaTeX, Excel, Tcl/tk, etc

Benefits

- Abstract away unnecessary details
- Improve
 - Performance
 - Reliability, maintainability ... = productivity

A functional MSP language can be particularly useful for implementing DSLs

Syntax in (Meta)OCaml

```
type exp =
```

```
    Int of int                | Var of string  
| App of string * exp        | Add of exp * exp  
| Sub of exp * exp          | Mul of exp * exp  
| Div of exp * exp          | Ifz of exp3
```

```
type def =
```

```
    Declaration of string * string * exp
```

```
type prog = Program of def list * exp
```

Syntax in (Meta)OCaml

```
Represent  let rec f x =  
            if x=0 then 1 else x*(f(x-1))  
          in f 10
```

```
As let termFact = Program  
    ([Declaration ("f", "x", Ifz(Var "x",  
        Int 1, Mul(Var "x", (App ("f",  
        Sub(Var "x", Int 1))))))],  
    App ("f", Int 10))
```

Syntax in (Meta)OCaml

```
Represent  let rec f x =  
            if x=0 then 1 else x*(f(x-1))  
          in f 10
```

```
As let termFact = Program  
    ([Declaration ("f", "x", Ifz(Var "x",  
        Int 1, Mul(Var "x", (App ("f",  
        Sub(Var "x", Int 1))))))],  
    App ("f", Int 10))
```

The natural benchmark

Note that we can type in what we want to represent directly into OCaml and run it:

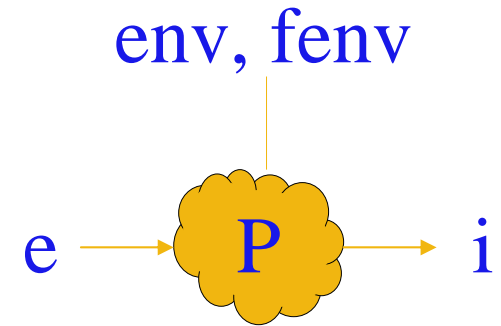
```
let rec f x =  
    if x=0 then 1 else x*(f(x-1))  
in f 10
```

This is NOT a generic solution.

But IT IS a natural benchmark for what constitutes good performance.

Interpreter

Runtime environments:



```
(*      env : string -> int
      fenv : string -> (int -> int) *)
```

```
exception Yikes
```

```
let env0 = fun x -> raise Yikes
```

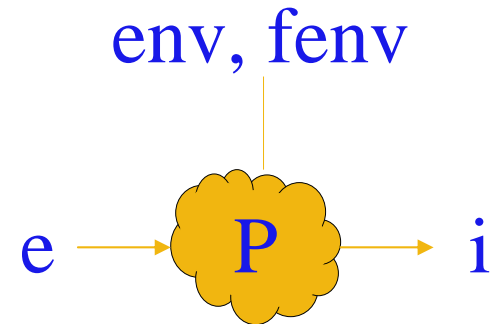
```
let fenv0 = env0
```

```
let ext env x v =
```

```
  fun y -> if x=y then v else env y
```

Interpreter

A safe interpreter for just expressions:



```
let rec eval3 e env fenv = match e with
```

```
  Int i -> Some i ...
```

```
| Div (e1,e2) ->
```

```
(match (eval3 e1 env fenv,
```

```
        eval3 e2 env fenv) with
```

```
  (Some x, Some y)->
```

```
    if y=0 then None else Some (x/y)
```

```
| _ -> None) ...
```

Interpreter

Same for programs/declarations

```
let rec peval3 p env fenv =
```

```
match p with
```

```
  Program ([],e) -> eval3 e env fenv
```

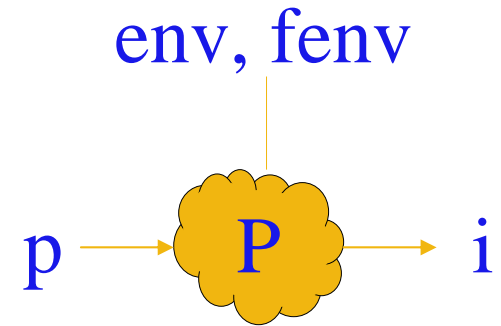
```
| Program (Declaration (s1,s2,e1)::t1,e) ->
```

```
  let rec f x =
```

```
    eval4 e1 (ext env s2 x)
```

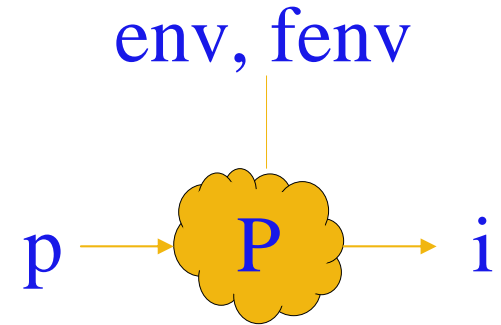
```
        (ext fenv s1 f)
```

```
  in peval3 (Program(t1,e)) env ...
```



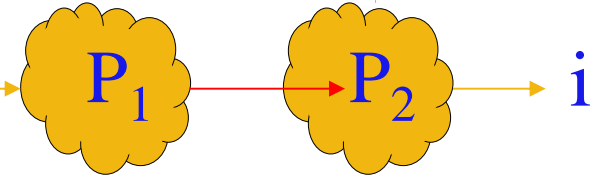
Interpreter

- The interpreter is generic
 - It will evaluate any program at runtime (even ones provided by the user)
- The interpreter is simple and maintainable
 - It is essentially a specification of the semantics of the language
 - (Actually, it's almost a denotational semantics)
- It runs 21x slower than our benchmark
 - Costly generic solutions are impractical, forcing us to resort to ad hoc solutions...



Staged Interpreter

We can traverse the program early: $e \rightarrow P_1 \rightarrow P_2 \rightarrow i$



```
let rec eval4 e env fenv = match e with
```

```
  Int i -> <Some i> ...
```

```
| Div (e1,e2) ->
```

```
  <(match (~(eval3 e1 env fenv),
```

```
    ~(eval3 e2 env fenv))
```

```
  with (Some x, Some y)->
```

```
    if y=0 then None else Some (x/y)
```

```
  | _ -> None) ... >
```

Staged Interpreter

Same for programs:

```
let rec peval4 p env fenv =
```

```
match p with
```

```
  Program ([],e) -> eval4 e env fenv
```

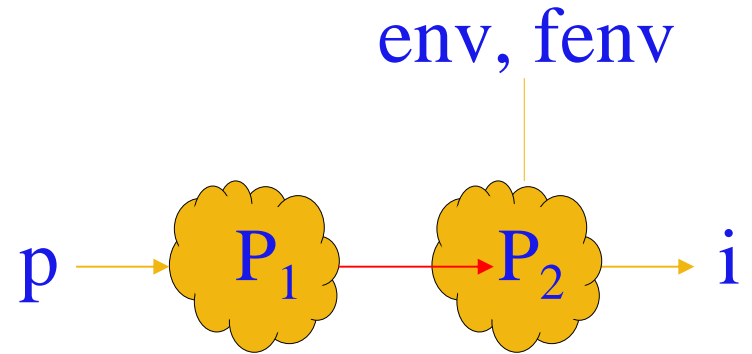
```
| Program (Declaration (s1,s2,e1)::t1,e) ->
```

```
  <let rec f x =
```

```
    ~(eval4 e1 (ext env s2 <x>)
```

```
      (ext fenv s1 <f>))
```

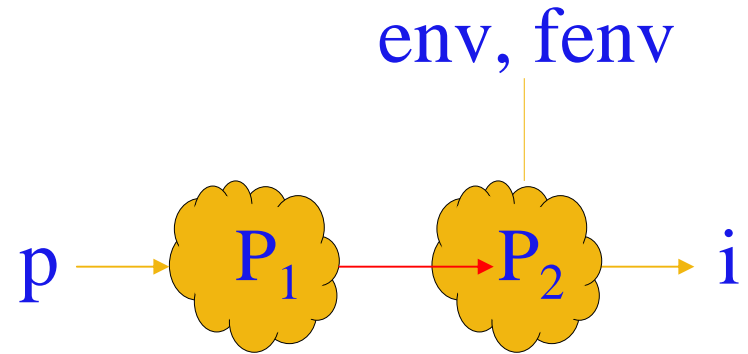
```
  in ~(peval4 (Program(t1,e)) env ... >
```



Staged Interpreter

- Staging helps

- We get code 4x faster than the original interpreter
- But that means it's 5x slower than our benchmark...



- We would like to generate the following code:

```
<let rec f x =  
    if (x = 0) then 1 else (x*(f (x-1)))  
in (f 10)>
```

- But what does it actually generate?

Staged Interpreter

- Generated code:

```
<let rec f x =
```

```
  match Some x with
```

```
    Some x -> if x = 0 then Some 1 ...
```

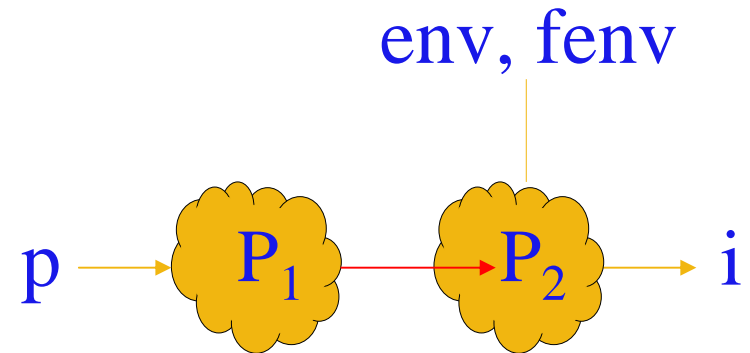
```
  | None -> None
```

```
in match Some 10 with
```

```
  Some x -> f x
```

```
  | None -> None>
```

- What happened?



Staged Interpreter

- Generated code:

```
<let rec f x =
```

```
  match Some x with
```

```
    Some x -> if x = 0 then Some 1 ...
```

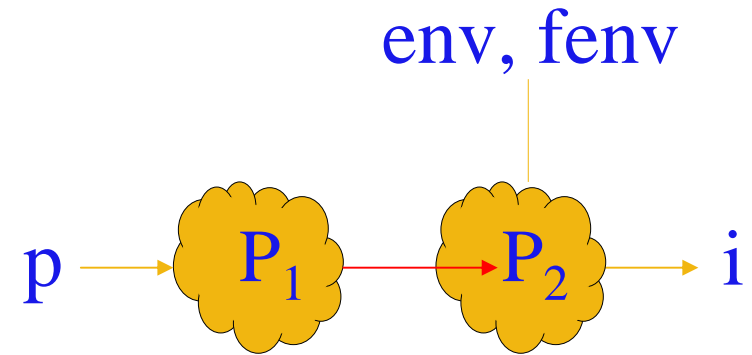
```
  | None -> None
```

```
in match Some 10 with
```

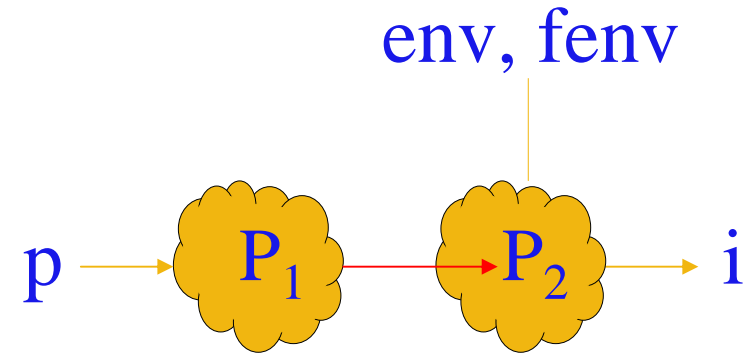
```
  Some x -> f x
```

```
  | None -> None>
```

- We are tagging and untagging...



Staged Interpreter

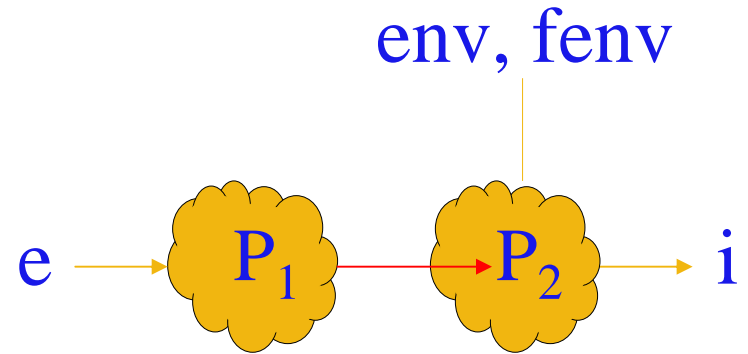


If there is ONE thing you want to take from this tutorial:

Avoid the temptation to find ways to post-process the code you generate. It can be arbitrarily better (in terms of performance and quality) to avoid generating bad code in the first place.

Staged Interpreter

The source of the problem:



```
let rec eval4 e env fenv = match e with
```

```
  Int i -> <Some i> ...
```

```
| Div (e1,e2) ->
```

```
  <(match (~(eval3 e1 env fenv),
```

```
          ~(eval3 e2 env fenv))
```

```
    with (Some x, Some y)->
```

```
      if y=0 then None else Some (x/y)
```

```
    | _ -> None) ... >
```

Interpreter (in CPS)

```
let rec eval5 e env fenv k =
```

```
match e with
```

```
  Int i -> k (Some i) ...
```

```
| Div (e1,e2) ->
```

```
  eval5 e1 env fenv (fun r ->
```

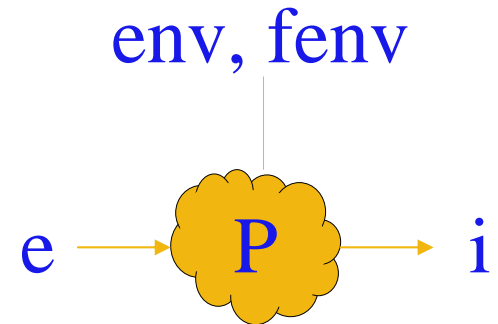
```
  eval5 e2 env fenv (fun s ->
```

```
  match (r,s) with
```

```
    (Some x, Some y) ->
```

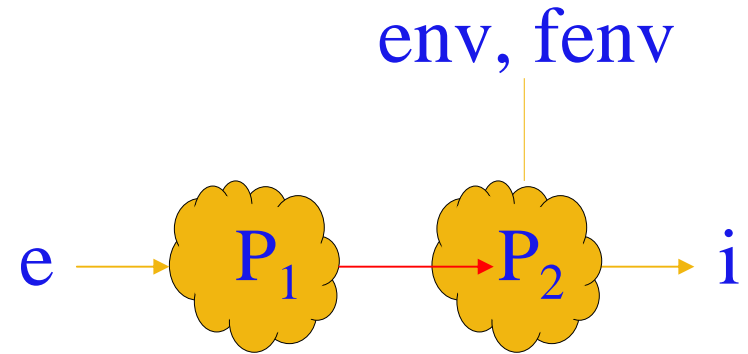
```
      if y=0 then k None else k (Some (x/y))
```

```
    | _ -> k None)) ...
```



Staged Interpreter

```
let rec eval6 e env fenv k =  
  match e with  
    Int i -> k (Some <i>) ...  
  | Div (e1,e2) ->  
    eval6 e1 env fenv (fun r ->  
      eval6 e2 env fenv (fun s ->  
        match (r,s) with  
          (Some x, Some y) ->  
            <if ~y=0 then ~(k None)  
              else ~(k (Some <~x / ~y>))>  
        | _ -> k None))
```



What we get

- The generated code is not exactly what we want:
 - Error check is inserted only where it is needed
 - Example:

```
<let rec f x =  
    if x=0 then 1 else x*(f (x-1))  
    in if y=0 then raise (Div_by_zero)  
        else f (20/y)>.
```

What about things like aspects?

- How can we implement the following language?

```
type aop = Prog of prog
          | Aspect of string * cmd * aop
type cmd = Ret
          | Time
          | If0 of exp * cmd * cmd
let p =
  Aspect("fib",
        If0(Sub(Var "#1", Int 3), Time, Ret),
        Prog fibFive)
```

Weaving interpreter

```
let rec weval cmd s1 env f =  
  match cmd with  
  | Ret -> f ()  
  | Time -> Trx.timenew  
    (s1^"("^(string_of_int (env "#1"))^")") f  
  | If0 (e,c1,c2) ->  
    if (eval e env fenv0)=0 then  
      weval c1 s1 env f  
    else  
      weval c2 s1 env f
```

Change the program interpreter

```
let rec peval p env fenv aenv =
  match p with
  | Program ([],e) -> eval e env fenv
  | Program (Declaration (s1,s2,e1)::tl,e) ->
    let rec f x =
      (weval (aenv s1)
        s1
        (ext env "#1" x)
        (fun () ->
          eval e1 (ext env s2 x) (ext fenv s1 f)))
    in peval (Program(tl,e)) env (ext fenv s1 f) aenv
```

Declaration interpreter

```
let rec aeval' a aenv =  
  match a with  
  | Prog p -> peval p env0 fenv0 aenv  
  | Aspect (s,c,a) -> aeval' a (ext aenv s c)  
  
let aeval a = begin Trx.init_times;  
                Format.printf "%d\n"  
                  aeval' a (fun x -> Ret);  
                Trx.print_times ()  
              end
```

Stage it, and you get:

```
# aeval' p env0;;
- : ('a, int) code =
.<let rec f x =
  let z = fun () ->
    if x=0 then 0 else if (x - 1) = 0 then 1 else f(x-1) + f(x-2)
  in if (x-3)=0 then Trx.timenew "fib"^(string_of_int x)^" z
    else z() in
  f 5>.
# aeval p;;
5
___ f(3) _____ 1048576x avg= 1.658149E-03 msec
___ f(3) _____ 1048576x avg= 1.634722E-03 msec
- : unit = ()
```

Summary of bigger example

- Language implementation is a “killer app”
- Staging annotations alone are not always enough. BTI’s are often needed.
 - CPS can be a huge help
 - Generated code has no interpretive overhead
 - See “Gentle Intro” for more details on “lint”
- Implementing AOP languages seems easy with MSP
 - This is an interesting new direction

Overview of this presentation

1. Staging the exponentiation function
 - A naïve model for writing MSP programs
 - Two steps: 1) write program, 2) stage it
 - Basic issues: Binding time, timing, termination, in-lining, code explosion
2. Staging a Markov chain problem
3. Staging a little interpreter (`lint`)
 - Binding time improvements. Adding AOP
4. Summary and “to probe further” pointers

Summary

Multi-stage programming languages:

- Capture the essence of “mixed computation”
 - Can provide a substitute for “macros”, “partial evaluation”, “program generation”, and “RTCG”
- Programmer can **see** when things are done
 - Means we know why our programs are faster
- Staging often requires only *minor annotations*
 - Means our programs are still easy to maintain
- Enjoy *sound and simple reasoning principles*

Summary (Cont'd)

- Hygienic expansion
 - Bound variables do not get accidentally captured
- Syntax and type safety
 - For a large class of programs, we are ***guaranteed before we generate anything*** that any generated program will compile successfully, and will run without runtime errors.
- All in all, provides radically new possibilities for language design and software development

Other applications in MetaOCaml

- Dynamic programming
 - Standard algorithms from CLR algorithms book
 - Generated code beats handwritten C implementation
- Fast-Fourier Transform
- Some more will be presented tomorrow at MetaOCaml Workshop
 - An implementation of Turtle Graphics
 - Web-page generation
 - Numerical computation

Results from research on MSP

- Design principles [PEPM '97], [TCS '00]
- Type system using “classifiers” [POPL '03]
- Equational theory [PEPM '00]
- Tag elimination [PADO '01], [ICFP '02a]
- Generative macros [ICFP '02b], [GPCE '03]
- Type system using “classifiers” [POPL '03]
- Comparison with TH, C++ [DSPG '04]
- Generating FFT circuits [EMSOFT '04]

Results from research on MSP

- Design principles [PEPM '97], [TCS '00]
- Type system using “classifiers” [POPL '03]
- **Equational theory [PEPM '00]**
- Tag elimination [PADO '01], [ICFP '02a]
- Generative macros [ICFP '02b], [GPCE '03]
- Type system using “classifiers” [POPL '03]
- Comparison with TH, C++ [DSPG '04]
- Generating FFT circuits [EMSOFT '04]

Equational reasoning [PEPM'00]

Not just informal claims (which are often *wrong*)

- Formally verified reduction & equality rules
 - Surprisingly simple
 - Sets are indexed, but reductions are not
 - Rules also apply to future-stage code (!)

Underlying theory. Proof

- Based on confluence and standardization
- Uses [Takahashi '91], generalizes [Plotkin '75]

Reasoning for the λ -calculus

Expressions and values:

$$e ::= i \mid x \mid e e \mid \lambda x.e$$

$$v ::= i \mid \lambda x.e$$

Reductions:

$$(\lambda x.e_1) e_2 \rightarrow_{\beta} e_1[x:=e_2]$$

Example Computations:

$$((\lambda y.\lambda x.y) 5) 6 \rightarrow (\lambda x.5) 6 \rightarrow 5$$

Computation:

- A sequence of reductions ending with a value

Problem: β breaks

Naïve application of β rule is **wrong**.

Consider the following simple examples:

$$(\lambda x.7) \sim e \rightarrow 7$$

$$(\lambda x.\sim x) \sim e \rightarrow \sim \sim e$$

$$\sim \langle e \rangle \rightarrow e$$

This was not known for a some time ('96-'00)

e.g. [Davies, Pfenning '96]

Annotated-term approach ('95-'99) too complex

[Glueck et al '95], [ICALP '98]

A multi-stage λ -calculus [PEPM'00]

Don't use one set of terms to define reductions:

$$e ::= i \mid x \mid e e \mid \lambda x.e \mid \langle e \rangle \mid \text{run } e \mid \sim e$$

Use *families* that *stratify* terms (and values):

$$e^n ::= i \mid x \mid e^n e^n \mid \lambda x.e^n \mid \langle e^{n+1} \rangle \mid \text{run } e^n \mid \sim e^{n-1}$$

$$v ::= i \mid \lambda x.e^0 \mid \langle e^0 \rangle$$

Reductions: $(\lambda x.e_1^0) e_2^0 \rightarrow_{\beta} e_1^0[x:=e_2^0]$

$$\text{run } \langle e^0 \rangle \rightarrow_{\text{R}} e^0$$

$$\sim \langle e^0 \rangle \rightarrow_{\text{E}} e^0$$

The family $\{e^n\}$ “slices” the set $\{e\}$.

What goes in what set

- $\{e^0\}$ has elements
 - Like: x , $\lambda x.x$, $\langle \lambda x.x \rangle$, $\langle \lambda x.\sim(f \langle x \rangle) \rangle$
 - But not like: $\sim x$, $\langle \lambda x.\sim \sim(f \langle x \rangle) \rangle$
- $\{e^1\}$ has elements
 - Like: $\sim x$, $\langle \lambda x.\sim \sim(f \langle x \rangle) \rangle$
 - But not like: $\sim \sim x$, ...
- $\{e^{n+1}\}$ includes $\{e^n\}$

Resources (-friendly :)

- MetaOCaml distribution
- MetaOCaml-users mailing lists
- Paper: “A gentle introduction to MSP”
 - “lint” example studied in detail (inc. inlining)
- MetaOCaml Workshop (tomorrow)
- Rice COMP 511 page

Key “multi-stage” people



Zino Benaissa



Tim Sheard



Emir Pasalic



Eugenio Moggi



Cristiano Calcagno



Xavier Leroy



Ed Pizzi



Amr Sabry



Steven Ganz



Michael Florentin



Patricia Johann



Roumen Kaiabachev



Hongwei Xi



Stephan Ellner



Kedar Swadi