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=<2*4>**	**a=<2*4>**
Escape:	**b=<9+~a>**	**b=<9+2*4>**
Run:	**c= .! b**	**c=17**

(Simplified) typing rules:

Term	Type	Term	Type
X	**T**	**<X>**	**<T>**
X	**<T>**	**~X**	**T**
X	**<T>**	**.! X**	**T**

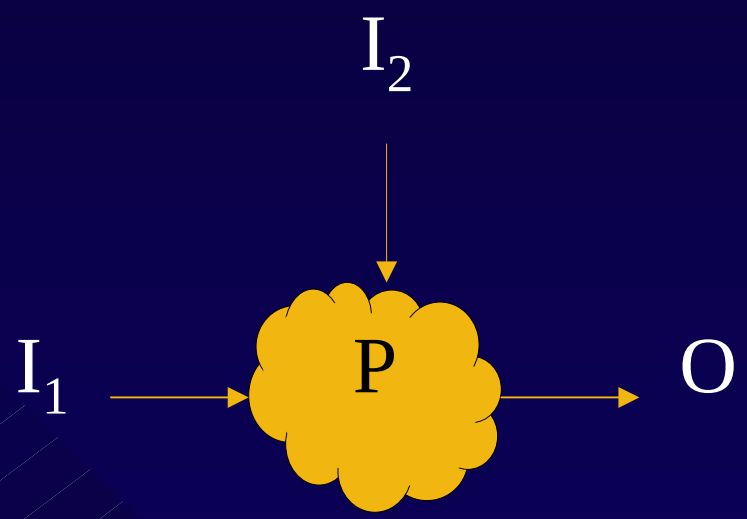
What does this mean?

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)

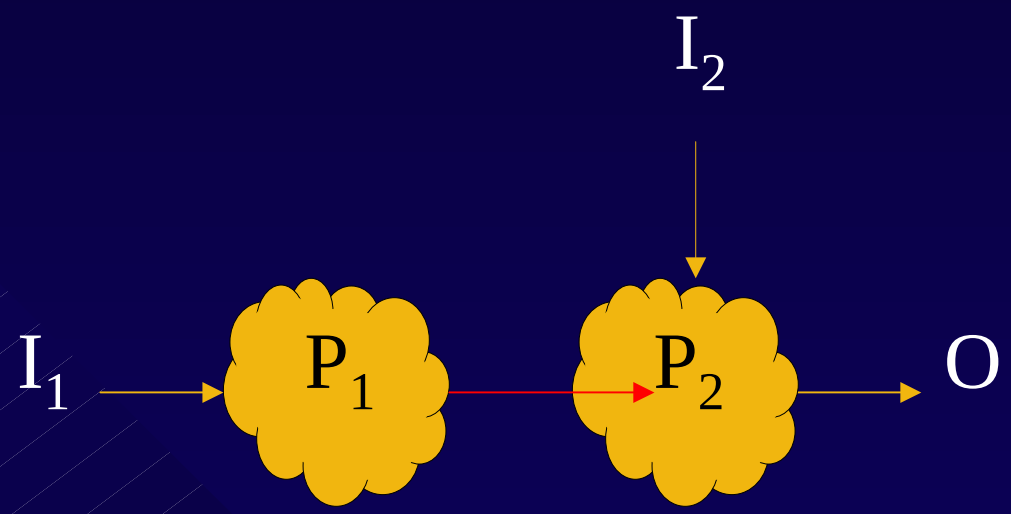
Common goal: ***Alter order of evaluation to reduce cost of a computation***

The abstract view



Batch Programming
Ex.: Interpreter

The abstract view



Multi-Stage Programming (MSP)
Ex.: Compiler

Where can MSP be used?

More places than we can count here. Examples:

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

Altering order of evaluation is a form of compilation

Overview of this presentation

1. Small example
   - 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. Cool example
3. Bigger example
   - A practical issue: Binding time improvement
4. Summary and “to probe further” pointers

Overview of this presentation

1. Small example
   - 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. Cool example
3. Bigger example
   - A practical issue: Binding time improvement
4. Summary and “to probe further” pointers

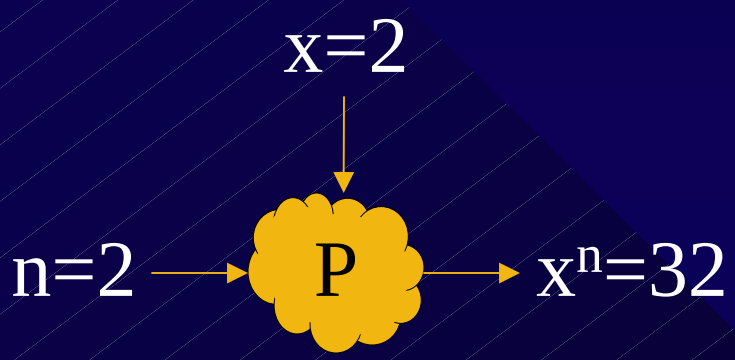
Small example: Exponentiation

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

Minimal example of a generic program.

Such a program is almost never used in practice.

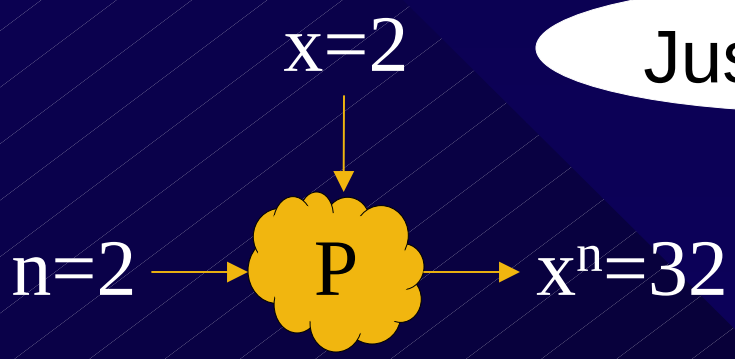
Why?



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

$$x^0 = 1$$

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

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

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

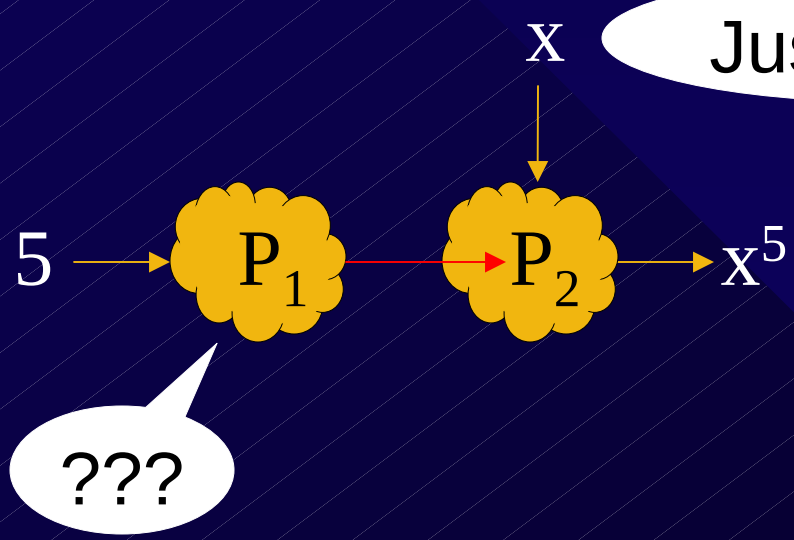
$= x * \underline{x^4}$

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

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

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

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

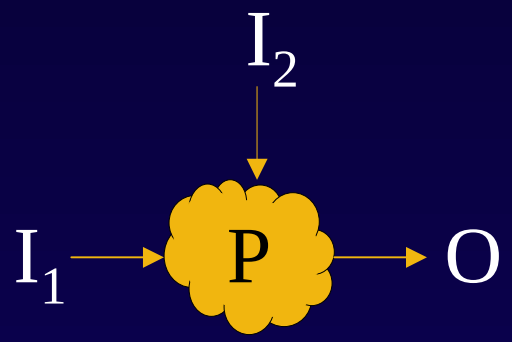


Just 4 ops!

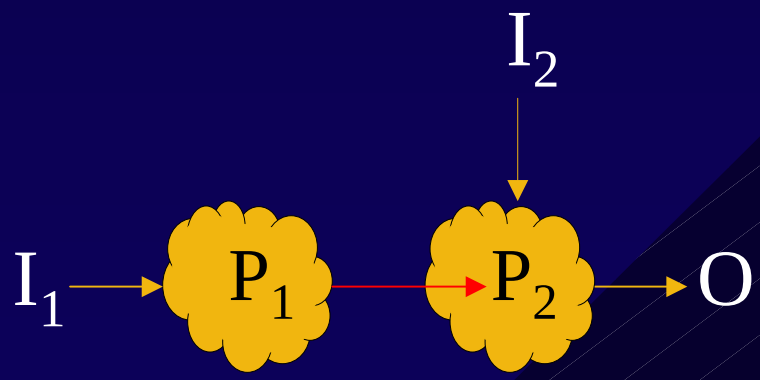
No recursion!

A Naïve MSP method

1. Write traditional, single-stage program



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



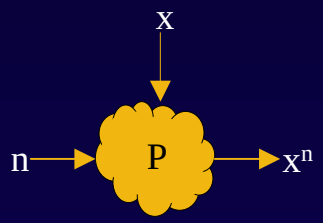
MetaOCaml

[GPCE '03]

- Builds on the OCaml system
  - Very successful, widely used, typed language
  - Compiled (both byte-code and native code)
    - Has competitive performance (esp. numerical apps)
    - Allows the collection of credible performance numbers
  - Substantial existing code base
- Realized by a source-to-source translation
  - Very close to an elegant formal model
  - Provides clear underlying cost model
- Details presented in GPCE talk tomorrow at noon

Small example: Exponentiation

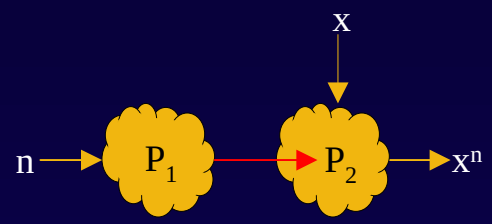
Single stage:




```
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));
```

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 gold are left for second stage

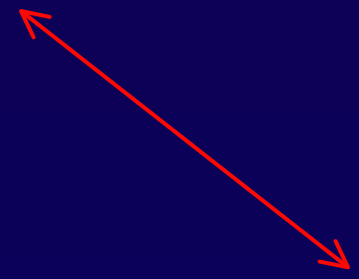
Small example: Exponentiation

To use the staged `exp` we simply write:

```
run <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 us = (* For unstaged version *)
      Trx.timenew “Unstaged” (fun () -> f a b);;
let s1 = (* Generation:  First part of stage 1 *)
      Trx.timenew “Stage 1” (fun () -> f’ a);;
let c2 = (* Compilation: Second part of stage 1 *)
      Trx.timenew “Compile” (fun () -> .! (c2 a));;
let s2 = (* Stage 2:  Applying a compiled function *)
      Trx.timenew “Stage 2” (fun () -> c2 b);;
Trx.print_times ();; (* Prints timing report *)
```

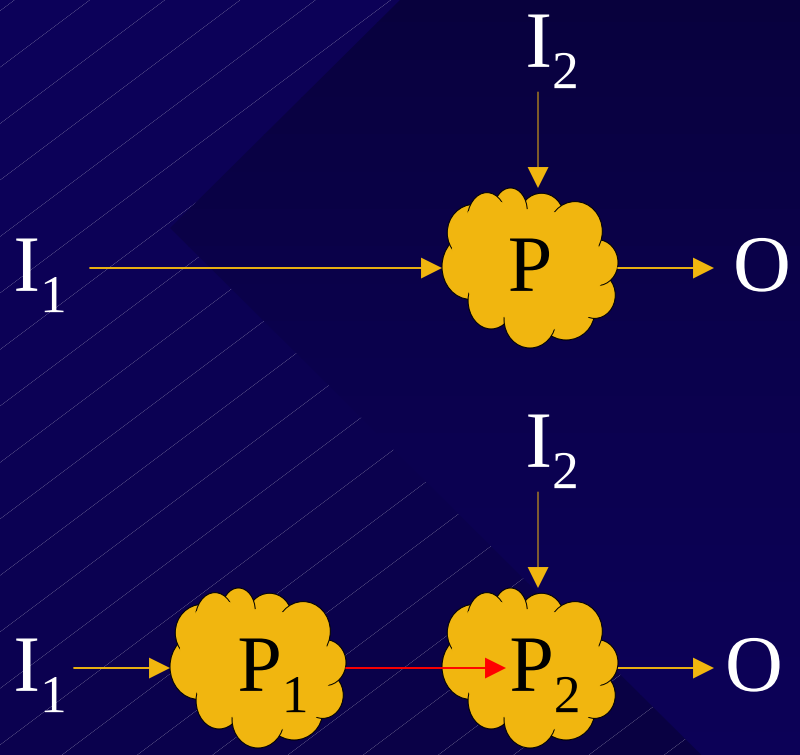
Timing in MetaOCaml (Careful!)

```
# #use "mex/3.ml";;  (* Load third example (power) *)
. . .
__ unstaged running _______ 100000x avg= 2.400000E-03 msec
__ stage 1 running ________ 100000x avg= 5.670000E-02 msec
__ compiling ______________________ 10x avg= 2.000000E+00 msec
__ stage 2 running ________ 100000x avg= 3.000000E-04 msec
-  : unit = ()
```

From this we can compute:

"Speedup" = (2.4E-3) / (3.0E-4) = 8 x

Typical speedups (MetaOCaml)



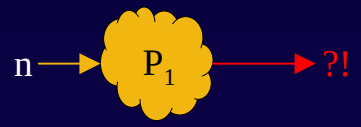
Program	Speedup $T(P)/T(P_2)$
Exp	5…8 x
Interp(fib 15)	4 x
Interp'(fib 15)	44 x
RewriteCPS	13 x
Chebyshev	3 x

What kinds of programs can we stage?

MSP languages allow the programmer to ***see (and check)*** what can be staged.

“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

In-lining using MSP

Instead of getting

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

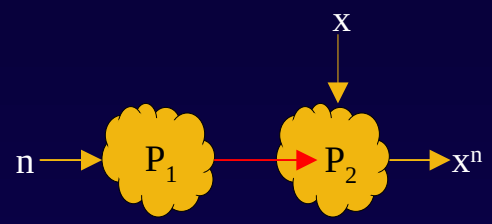
It would be nice if we can get

```
<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>
```

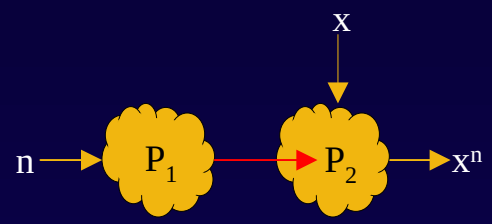
Note change in type:

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 gold

In-lining using MSP

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

we get code explosion: We generate

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

Solution (for 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)>
```

Common source of confusion: Partial *Evaluation* vs. *Application*

"Curried application" is latter, NOT the former.

For example: `F(x,y,z)=` `x+y+z`

Is same as: `G(x,y)=` `fun z ->` `x+y+z`

Now `G(1,2)` evaluates to `fun z -> 1+2+z`

but NOT: `fun z -> 3+z`.

To do the last step, we really need MSP.

Capture is a non-problem in MSP

Given the following definitions

```
let t(y)= <fun x-> x + ~y>
let p   = <fun x-> ~(t <x>) 10>
```

We do NOT get

```
    p   = <fun x->  (fun x-> x+x)10>
```

But rather

```
    p   = <fun x1-> (fun x2-> x2+x1)10>
```

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 …>`
- Hard: `<let f(x)= ~~(run <x>) in …>`

  → `<λx.~x>`

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

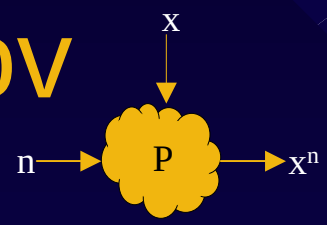
Summary of small example

- Staging make writing generic programs that don't pay a runtime overhead possible
- Staging annotations are lightweight
- Gathering performance timing in MetaOCaml
- Common pitfalls and how to deal with them
  - Termination, code explosion, variable capture (non-issue), type safety
- Why staging is different from partial application

Overview of this presentation

1. Small example
   - 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. Cool example
3. Bigger example
   - A practical issue: Binding time improvement
4. Summary and “to probe further” pointers

Cool example: Simple Markov



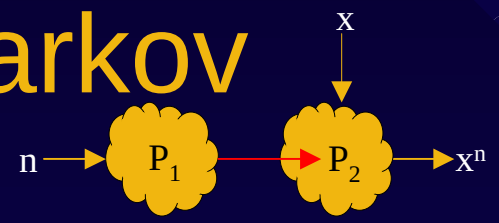

```
let mkmatrix rows cols p =
  let last_col = cols - 1
  and m = Array.make_matrix rows cols 0.0
    in
  for i = 0 to rows - 1 do
    let mi = m.(i) in
    for j = 0 to last_col do
     let r  = rows / 4 in
     let mn = max 0 (i-r) in
     let mx = min rows (i+r) in
     let rn = mx-mn-1 in
     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);;
```

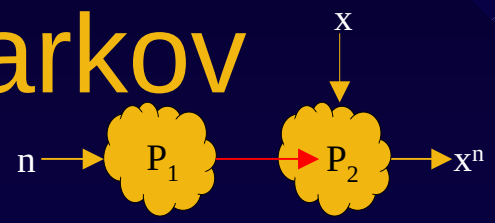
Cool example: Simple Markov



Staged model construction:

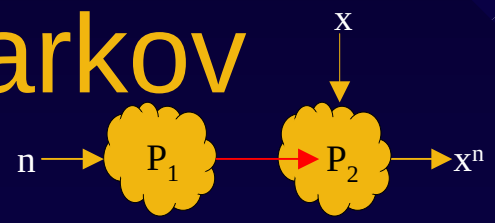
```
let lift x = <x>;;          (* This is a non-trival function *)
let mkmatrix rows cols p =
  let last_col = cols - 1
  and m = Array.make_matrix rows cols <0.0> in
  for i = 0 to rows - 1 do
    let mi = m.(i) in
    for j = 0 to last_col do
     let r  = rows / 4 in          let mn = max 0 (i-r) 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
```

Cool example: Simple Markov




```
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))>. ;;
```

Cool example: Simple Markov



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
```

Speedup = 1.4E3 / 6.5E-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 exists, but not yet in implementation.

Summary of cool example

- Speedups can be arbitrarily large
  - But remember, “speedup” is not the point
- MSP can be useful for numerical applications
- Imperative programs, too, can be staged
- Current type system allows scope extrusion
- Lift: a staging pattern, and when to use it
- Type system doesn’t prevent scope extrusion

Overview of this presentation

1. Small example
   - 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. Cool example
3. Bigger example
   - A practical issue: Binding time improvement
4. Summary and “to probe further” pointers

Domain Specific Languages (DSLs)

Examples

- Lex, yacc, HTML, PERL, Tkl/tk, etc

Benefits

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

A functional MSP language is a particularly useful for building 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

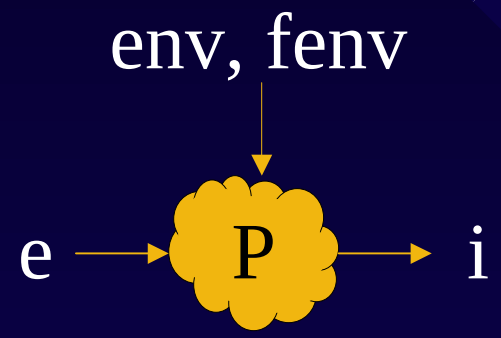
Note that we can type in what we want to represent directly into OCaml and run it:

```
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.

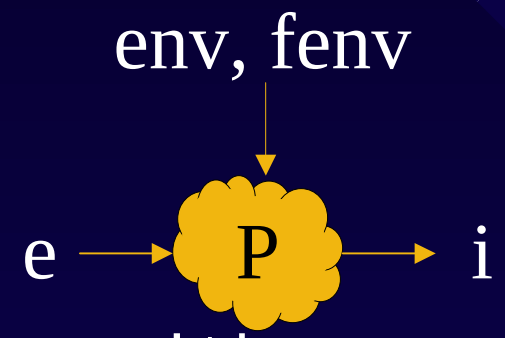
Interpreter



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
```

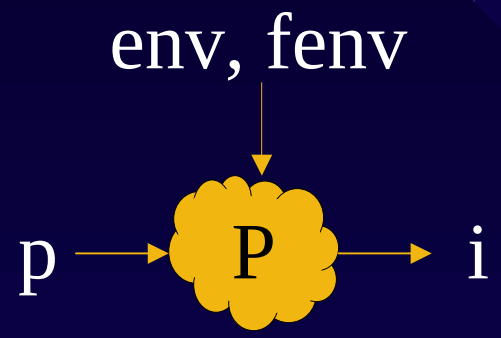
Interpreter



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) ...
```

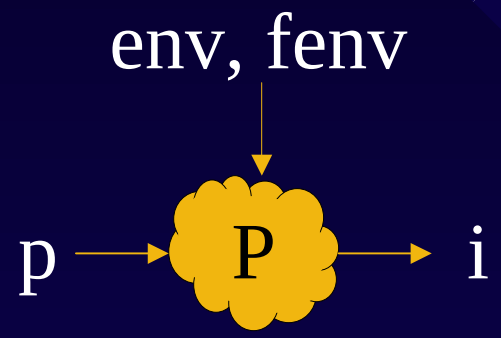
Interpreter



Same for programs/declarations

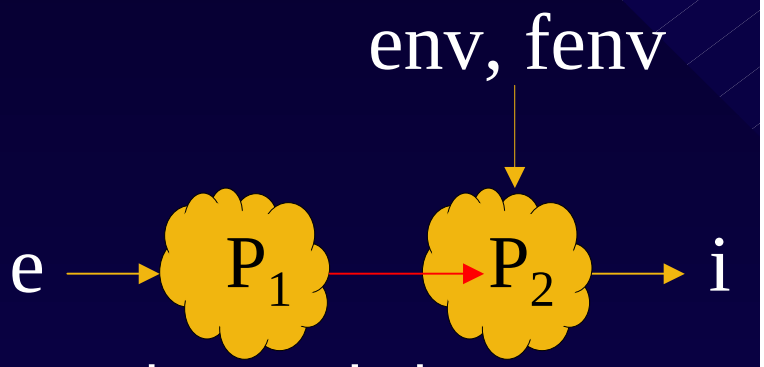
```
let rec peval3 p env fenv=
match p with
 Program ([],e) -> eval4 e env fenv
|Program (Declaration (s1,s2,e1)::tl,e) ->
 let rec f x =
   eval4 e1 (ext env s2 .<x>.)
            (ext fenv s1 .<f>.)
   in peval4 (Program(tl,e)) env ... >.
```

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
  - Such a high-cost for this generic solution could make impractical, forcing us to resort to ad hoc solutions…

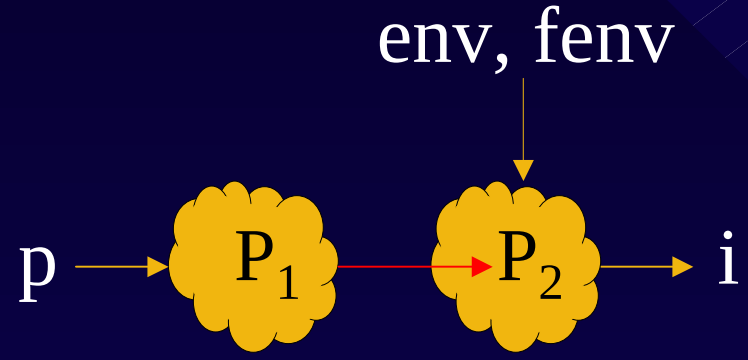
Staged Interpreter



We can traverse program early:

```
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) ... >
```

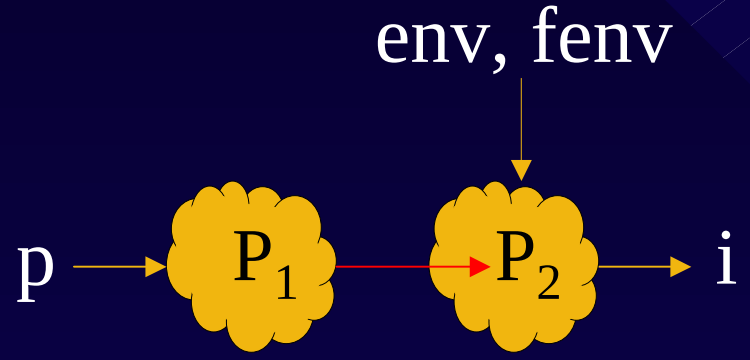
Staged Interpreter



Same for programs:

```
let rec peval4 p env fenv=
match p with
 Program ([],e) -> eval4 e env fenv
|Program (Declaration (s1,s2,e1)::tl,e) ->
 .<let rec f x =
   .~(eval4 e1 (ext env s2 .<x>.)
               (ext fenv s1 .<f>.))
   in .~(peval4 (Program(tl,e)) env ... >.
```

Staged Interpreter

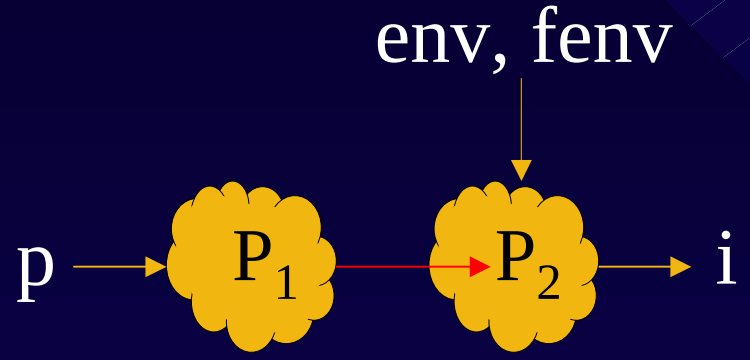


- Staging helps
  - We get code 4x faster than the original interpreter
  - But that means it's 5x faster than our benchmark…
- We would like to generate the following code:

```
.<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

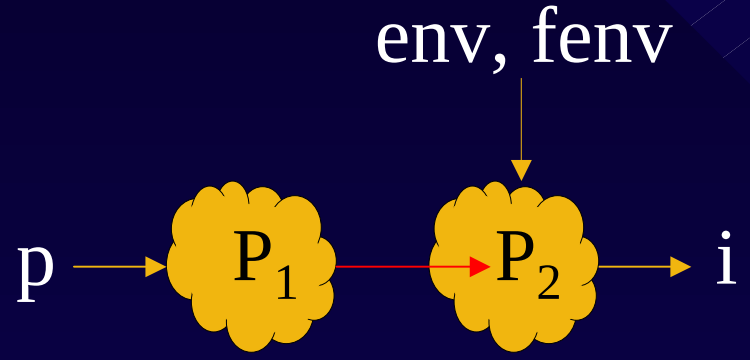


- Generated code:

```
.<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

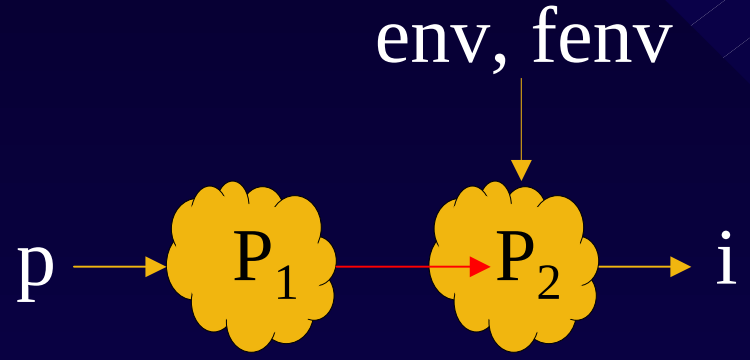


- Generated code:

```
.<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

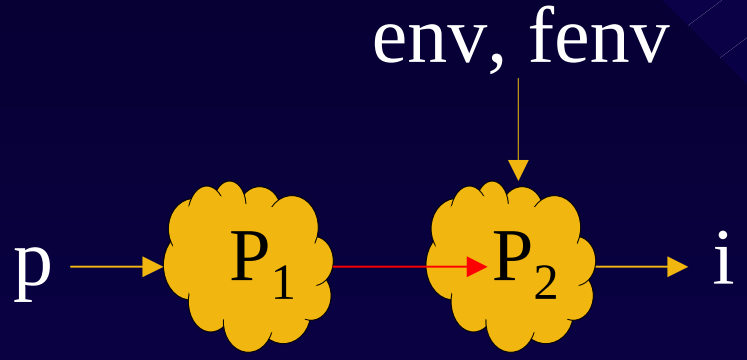


- Generated code:

```
.<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>.
```

- And it’s not even necessary

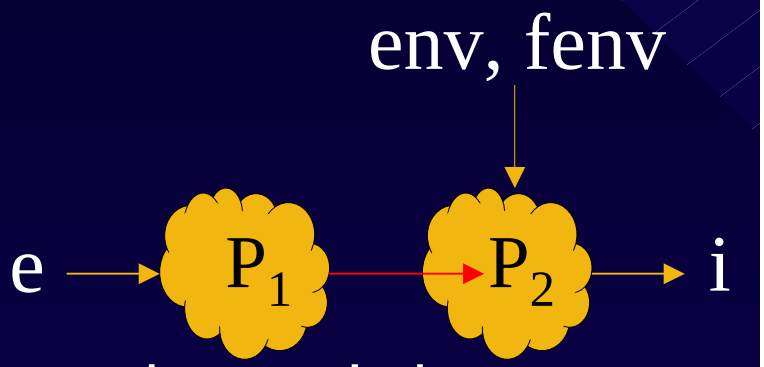
Staged Interpreter



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

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.

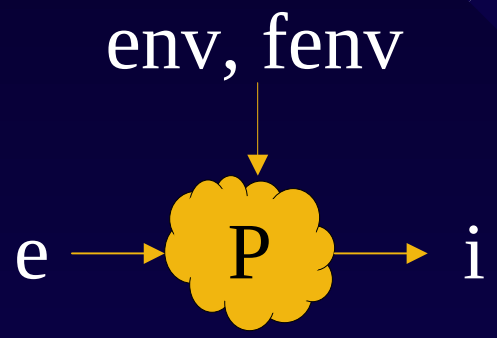
Staged Interpreter



The source of the problem:

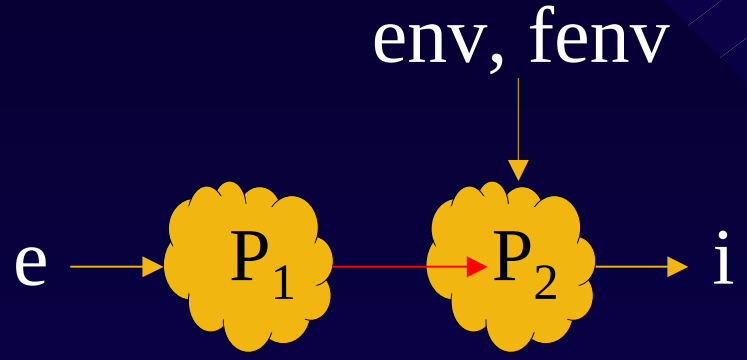
```
let rec eval4 e env fenv = match e with
  Int i -> <Some I> ...
| Div (e1,e2) ->
 <(match (~(eval3 e1 env fenv),
          ~(eval3 e2 env fenv)) with
    (Some x, Some y)->
      if y=0 then None else Some (x/y)
  | _ -> None) ... >
```

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))
```

Summary of bigger example

- Language implementation is a killer app for MSP
- Staging annotations alone are not always enough
- Often, binding-time improvements are needed
- Staging with CPS gets us there
  - We generate the code that we want (21x speedup)

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

- Can we beat this code? Yes! Details in paper

Overview of this presentation

1. Small example
   - 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. Cool example
3. Bigger example
   - A practical issue: Binding time improvement
4. Summary and “to probe further” pointers

Summary

Multi-stage programming languages:

- Capture the essence of “mixed computation”
  - Means we can speed up our programs
- 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
- Proven to enjoy a basic set of ***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

Results from research on MSP

- The design principles and applications of MSP

  [PEPM ’97], [TCS ’00]

- Developing first sound type system (& semantics)

  [ICALP ’98]

- Developing the notion of closed types (expressivity)

  [ESOP ’99],

  [ICALP ’00]

- Developing an equational theory, proving soundess

  [PhD ’99], [PEPM ’00]

- Tag elimination (a specialized optimization)

  [DTP ’01], [PADO-II ’01]

Results from research on MSP

- The design principles and applications of MetaML

  [PEPM '97], [TCS '00]

- Developing first sound type system (& semantics)

  [ICALP '98]

- Developing the notion of closed types (expressivity)

  [ESOP '99],

  [ICALP '00]

- Developing an equational theory, proving soundess

  [PhD '99], [PEPM '00]

- Tag elimination (a specialized optimization)

  [DTP '01], [PADO-II '01]

A multi-stage λ-calculus [PEPM'00]

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

- Formally verified "reasoning principles"
- Rules surprisingly simple
  - Sets are indexed, but reductions are not
  - Rules also apply to future-stage code (!)

The soundness proof itself

- Quite technical (confluence, 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 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:

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

$$\text{run } \langle e^0 \rangle \rightarrow_R e^0$$

$${\sim}\langle e^0 \rangle \rightarrow_E e^0$$

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

What goes in what set

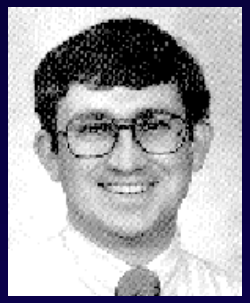
- $\{e^0\}$ has elements
  - Like: x, $\lambda x.x$, $< \lambda x.x >$, $< \lambda x.\sim(f <x>) >$
  - But not like: $\sim x$, $< \lambda x.\sim \sim(f <x>) >$
- $\{e^1\}$ has elements like
  - Like: $\sim x$, $< \lambda x.\sim \sim(f <x>) >$
  - But not like: $\sim \sim x$, …
- $\{e^{n+1}\}$ includes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

Resources (Google™-friendly :)

- MetaOCaml distribution
  - Native code compiler being tested, to be released soon
- MetaOCaml-users mailing lists
- Tutorial paper: “A gentle introduction to multi-stage programming” (also on tutorials CD)
  - Bigger ex. studied in detail + intro to research literature
- Multi-stage programming course page
  - COMP 511 at Rice: Research focus, bigger examples (term projects), current research topics

Key “multi-stage” people


Zino Benaissa


Tim Sheard


Emir Pasalic

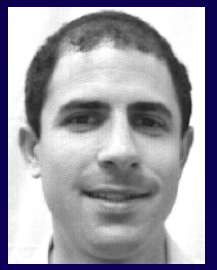

Eugenio Moggi


Cristiano Calcagno


Xavier Leroy


Ed Pizzi


Amr Sabry


Steven Ganz


Michael Florentin


Patricia Johann


Roumen Kaiabachev


Hongwei Xi


Stephan Ellner