# Partial Evaluation

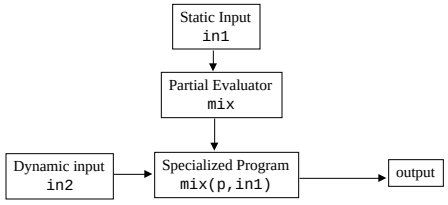
1. Introduction
2. Motivation, Examples
3. The First Futamura Projection.
4. The Second.
5. You guessed it , the third!
6. Simple Example 1: A Turning Machine Interpreter *TI* and the first Futamura projection with *TI.*
7. Under the hood-basics of partial evaluation.
8. Example 2: *mix* for a simple flowchart language.
9. Difficulties in doing it for C.
10. Tempo and applications.
11. Conclusion





# What is it?

A scheme for program execution:




```
f(n,x) = if n = 0 then 1
                  else if even(x) then (f(n/2, x))^2
                       else x * f(n-1,x)

mix(f,n=4) =???

mix(f,n=5) = ???

mix(f,x=5) = ???
```

*What kinds of computations are involved in deriving the specialized programs?*

Main Techniques: **Symbolic Computation, Unfolding, Program point specialization.**

How does one describe the partial evaluation process.

## Is Partial Evaluation worth the effort?

Worth it if:

*time(*`mix(p,in1)`*)*+
*N*time((*`mix(p,in1))(in2)`*)*
<
*N*time(*`p(in1,in2)`*)*

*Other Benefits (having your cake and eating it):*

- Write a very general-purpose program, to solve a class of problems (likely to be slow *why??)* and specialize it to a particular case (*printf?*)
- Write a very modularized program, with lots of control and data abstraction, separation of concerns etc (also slow. *why?).* And then *de*-modularize, *de*-abstract, and *unite* concerns--for efficiency. *But keep the long-suffering programmer* **out of it*!!***
- Inter-language translation, producing compilers from interpreters, and producing compiler-generators from partial evluators (Futamura projections, the holy grail of partial evaluation research).





## General Examples

- **Graphics:** Specializing a ray tracer with respect to scene..can be useful for *what?*
- **Strings:** Specializing a regular expression matcher with respect to a specific pattern...*what happens?*
- **Circuit Simulation:** how?
- **Cryptography:** Not much work but seems natural...*how?*
- **Database Querying:** *wrt query.*
- **Linear Programming:**

***Watch out for "Stupid `mix` tricks".***

Example:

```
f(n,x) = if n = 0 then 1
                 else if even(x) then (f(n/2, x))^2
                       else x * f(n-1,x)
mix(f, n = 5) =
(let n =5 in
   if n = 0 then 1
                 else if even(x) then (f(n/2, x))^2
                       else x * f(n-1,x))
```

## Futamura Projections

**Motivation:** Writing interpreters is easier than writing compilers. *Why?* (only one language to think of; only onetime; if written in a high-level language, can be declarative and modular (like operational semantics).

But.. compilers => Faster code + smaller runtime.

**So!** why not generate compilers from Interpreters?

**First,** some background notational devices:

$\|source\|_S$ = *Function denoted by L-language program* $source$

$\|source\|_S(\overline{in})$ = *Result of applying this function to input* $\overline{in}$

$tarrget = \|compiler\|_L(source)$ *Translate from S to T.*

$\|source\|_S(\overline{in}) = \|tarrget\|_T(\overline{in})$

$\|source\|_S(\overline{in}) = \|\|compiler\|_L(source)\|_T(\overline{in})$





## Futamura Projection (contd)

First, some consistent notation (as before) about how *mix,* the partial evaluator works (mix's output language??)

$\|source\|_S(\overline{in}) = \|source\|_S(\overline{in_1}, \overline{in_2})$

$source, in\ \|source\|_S(\overline{in}) = \|\|mix\|(source, \overline{in_1})\|(\overline{in_2})$

Now consider an interpreter $I$ for language $S$, written in language $T$.

$\|source\|_S(\overline{in}) = \|\|I\|_T(source, \overline{in})\|$

Interpreter $I$ has two inputs; let's specialize it with respect to one of it's inputs--*What language is the specialized versionof I??*

$\|source\|_S(\overline{in}) = \|\|mix\|(I, source)\|_T(\overline{in})$

$tarrget = \|mix\|(I, source)$

So we can use a partial evaluator with an interpreter, to compile!!

**First Futamura Projection:**

$$\boxed{tarrget = \|mix\|(I, source)}$$

## The Futamura Saga (contd)

$$\|source\|_S = \|\|mix\|(I, source)\|_T$$

*What does the result of the first FP do?*
It is a program, in *T*, which does what *I* would do, if it's input (the program it is interpreting) were *source.*

*How would the "stupid PE trick" be applied here, to compile source to Target?*

*What do interpreters look like?*

*How would an interpreter I be specialized as it is partially evaluated with respect to a source?*

*How would the speed of mix(I, source) compare with directly interpreting source?*

*What would happen if we had a partial evaluator for C, and we used it on the JVM, with a Java bytecode file?*

*What if we wanted to do this over and over again?*

$$tarrget = \|mix\|(JVM, source_2)$$

$$tarrget = \|mix\|(JVM, source_3)$$

.....





## The Second Futamura Projection

Since the first input is always the interpreter, why not specialize it?

$$tarrget = \|\|mix\|(mix, I)\|(source)$$

But recall:

$$tarrget = \|compiler\|(source)$$

So we get:

$$compiler = \|mix\|(mix, I)$$

*What does this compiler do?*
This is a program in the language that *mix* is written in, that does exactly what *mix* would do, if it's first input were an interpreter.

*What does this compiler look like?*
Very very different from a traditional compiler. We'll see an example.

*What is the performance of the compiliation process?*

*What are the requirements on "mix" to make this possible?*

*Can we make Java compilers for Suns, SGIs, x86.....*

## Working up to The Grand Finale

$$compiler = |mix\|(mix, JVM_{Sun})$$
$$compiler = |mix\|(mix, JVM_{SGI})$$
$$compiler = |mix\|(mix, JVM_{x86})$$
$$compiler = |mix\|(mix, PLAN_{x86})$$
$$.compiler = \|mix\|(mix, Smalltalk_{x86})$$
$$.....$$

*So many interpreters, so little time....*

Why not precompute (partially evaluate the *mix* itself, and specialize it for the task of compiler generation??

$$compiler = \|mix\|(mix, I)$$
$$compiler = \|\|mix\|(mix, mix)\|(I)$$
$$cogen = \|mix\|(mix, mix)$$





## The Third Futamura projection.

$$cogen = \|mix\|(mix, mix)$$

*What does this compiler generator look like?*

"Don't worry about it, it's not very intuitive" (from the book). *Surely they jest.*

*What does this compiler generator do, when given an interpreter I as input?*

It does what *mix* would do, if it's first input were *mix*, and it's second input were *I. What is that?*

*How do we evaluate it's peformance?*

## Some Observerations.

*Speed:*

1. Compiled code from *mix(I, source)*, should be faster than running *I* over *source.*
2. Compiling with *compiler* (*mix(mix, I))* should be faster than running *mix(I, source).*
3. Generating compilers with *cogen (mix(mix,mix))* should be faster than running *mix(mix,I)).*

*(from book:)* Each application boosts speed x10!

*Self-Applicability:*

In order for all this to work, the *mix* has to be self-applicable. This is a **tricky business**. *Why is that?*

*Usability:*

1. *mix* operation should be autonomous.
2. Resulting specialization should not have to be manually "post-processed" or even looked at.
3. *mix* should be a "black box"
4. *mix* should predict/describe performance of specialization.
5. *mix* should allow debugging/testing using familiar methods.

*Why are some of these (which?) requirements hard to realize?*





## More Observations.

*Analogy with parser generation:*

*Yacc:* not all CFGs can be processed; need to eliminate various conflicts (shift-reduce, reduce-reduce).

*Mix:* not all programs can be specalized; need to be amenable to *binding-time separation.*

*Cannot always specialize:*

e.g., x^n in previous example cannot be specalized from knowing x.

## A simple Example.

*A program fragment:*

```
1 while (name != hd(namelist)) do
2 begin
3       valuelist := tl(valuelist); /* tl=tail*/
4       namelist := tl(namelist)
6 end
7 value := hd(valuelist);
```

*Assume inputs:*

```
name=z, namelist = (x,y,z).
```

*What is the result?*

```
3.1 valuelist=tl(valuelist);
3.2 valuelist=tl(valuelist);
7.1 value := head(valuelist);
```

*Why did line 4 go away?*

*Why are there 2 copies of line 3?*

*Why did the while loop go away?*

*What determines what goes away, and what stays?*

**Intuitions behind some jargon:**

"mixed computation" i.e., *mix.*
"binding time analysis"
"polyvariant specialization"
"go to compression"

**Next:** Actual example with $\mathcal{FP}$ 1, and $\mathcal{FP}$ 2!!!





## Example Interpreter: Turing Machine.

*Tape:*

$$....a_{-3}a_{-2}a_{-1}\underline{a_0}a_1a_2a_3.....$$

*Operation:*

Alphabet: *0,1,b.*
Input is *a0* thru *an.* All else blank (*b*)
*a0* is "scanned square"
Output is the tape to the Right of the head.

*Example Program P:*

```
0: if 0 goto 3
1: right
2: goto 0
3: write 1
```

*Example Input String:*

```
110101
```

*What is the output?*

*Some meta-ops for Interpreter:*

Instructions with lists: `hd, tl` etc.
`Left` and `Right` tapes represented with lists. '() for all blank tape section.
Need special version of `hd` for empty tape: `firstsym`
Also, `nthTail` for `goto's.`
Also `firstInst`

Examples???

## Turing Interpreter *TI* in a FC language.

```
read(Q, Right);
init:   Qtail := Q; Left:= '()
loop:   if Qtail = '() goto stop else cont;
cont:   Instruction := firstInst(Qtail);
        Qtail := rest(Qtail);
        Op := hd(tl(Instruction));
        if Op = 'right goto do-r  else cont1;
cont1:  if Op = 'left  goto do-l  else cont2;
cont2:  if Op = 'write goto do-w  else cont3;
cont3:  if Op = 'goto  goto do-g  else cont4;
cont4:  if Op = 'if    goto do-if else error;
do-r:   Left := cons(firstsym(Right), Left);
        Right:= tl(Right); go to loop;
do-l:   Right := cons(firstsym(Left), Right);
        Left:= tl(Left); go to loop;
do-w:   Sym := firstsym(tl(tl(Op)));
        Right:= cons(Sym,tl(Right)); go to loop;
do-g:   GoLabel:=hd(tl(tl(Instruction)));
        Qtail=nthTail(Q,GoLabel); go to loop;
do-if:  Sym:=hd(tl(tl(Instruction)))
        GoLabel:= hd(tl(tl(tl(tl(Instruction)))))
        If Sym = firstsym(Right) goto jump
           else goto loop;
jump:   Qtail :=nthTail(Q,GoLabel); goto loop;
error:  return("Syntax ERROR");
stop:   return Right;
```





## *mix(TI, P)*

*Original Program P:*

```
0: If 0 goto 3
1: right
2: goto 0
3: write 1
```

*Result of mix(TI,P)*

```
read(Right);

lab0:  Left := '();
       If '0 = firstsym(Right) goto lab2;
       else lab1;

lab1:   Left:=cons(firstsym(Right), Left);
        Right := tl(Right);
        If '0 = firstsym(Right) goto lab2;
        else lab1;

lab2:   Right = cons('1, tl(Right));
        return(Right);
```

*Are these two programs in the same language?*

*Do they mean the same thing?*

*Where did the structure of this program come from?*

*Where did the pieces of this specialized program come from?*

*What pieces of the interpreter are missing? why?*

17

## Under the Hood

*Some Definitions:*

**Computational State (*CS*):** a pair *(pp, store)* where *pp* is a location in a program, and *store* has current values of all the variables. Prog execution updates state. With only partial *store*, no execution, but can specialize.

**Static & Dynamic:** Static vars are known at specialization time. Classification into static/dynamic is known as **division.**

**Uniformly Congruent division:** at any transition from *(pp, store)* to *(pp',store')* all static variables at *store'* are computable from those at *store. i.e.*, at any assignment,

```
var := Exp
```

`var` must be dynamic if any variable in `Exp` is.


17


18

## Examples

*Program:*

```
1 while (name != hd(namelist)) do
2 begin
3           valuelist := tl(valuelist);
4           namelist := tl(namelist)
6 end
7 value := hd(valuelist);
```

*Program Points (variable names omitted):*

```
(1, [z,[x, y, z], _, [1,2,3]])
(4, [z,[x,y, z], _, [2,3]])
etc, etc.
```

*A division:* `([name,namelist], [value,valuelist])`

*Is this division uniformly congruent?*

i.e., at every assignment, can we calculate the values of all the static variables from the values before?


18

## More Definitions

**Specialized Program Point:** a tuple `(pp,vs)` consisting of a program point, and the values of the static variables.

**Polyvariant set:** the set of all specialized program points for a program, given some static variable values.

*Example Program:*

```
search: if (name != hd(namelist)) goto found;
        else goto cont;
cont:   valuelist := tl(valuelist);
        namelist := tl(namelist);
        goto search;
found   value:= hd(valuelist);
```

*Polyvariant set (with* `name=z, namelist=[x,y,z]` *):*

```
poly= [     (search,[z,[x,y,z]]),
     (search,[z,[y,z]]),(search,[z,[z]]),
     (cont, [z,[x,y,z]]),(cont,[z,[y,z]]),
     (found, [z,[z]])
     ]
```

*How did we compute this set?*

*What is this definition for?*

*Assume we have the polyvariant set, what do we do next?*





## Program Point Specalization.

| Command | Done at Specialization Time | Generated Code |
|---|---|---|
| `X:=Exp (X dynamic)` | `redex= reduce(Exp, vs)` | `X:=redex` |
| `X:=Exp (X static)` | `update(vs,x ,eval(Exp,v s)` | |
| `return Exp` | `redex= reduce(Exp, vs)` | `return redex` |
| `goto pp'` | `goto (pp',vs)` | |
| `if*, exp static, value true:` | | `goto (pp',vs)` |
| `if*, exp static, value false` | | `goto (pp'',vs)` |
| `if*, exp dynamic` | `redex= reduce(Exp, vs)` | `if redex goto(pp',vs ) else goto (pp'',vs)` |

*If statement format:*

**`if* exp goto pp' else goto pp''`**

## Computing the Polyvariants.

*Simple Specialization Algorithm:*

```
poly:= [(pp0,vs0)]
while unmarked (pp,vs) in poly do
begin
   mark(pp,vs)
   generate specialized code for the basic block starting pp using value vs
      following the table on previous slide.
   poly = poly U successors(pp,vs)
end
```

*How to compute* `successors(pp,vs)`*:*

Assume basic block starting at pp transforms state vs to state vs'

| Control Operator | Successors | Condition |
|---|---|---|
| return | [] | |
| goto pp' | [(pp',vs')] | |
| if* | [(pp',vs')] | exp static & exp -> true |
| if* | [(pp'',vs')] | exp static & exp -> false |
| if* | [(pp',vs'), (pp'',vs')] | exp dynamic |





## Comments on Mix Algorithm

*Each type of statement in the input language is handled according to it's semantics, doing some compuations and deferring others.*

*When values can be computed, they are, and store is updated.*

*If they cannot be, code gets generated.*

*Several, different(ly) specialized versions of an input program may be created. (example?)*

23

## Comments on Result of *mix(mix,TI)*

*The resulting compiler has the structure of TI.*

1. In the first FP, the resulting *target* had the structure of source.
2. Code generation not in syntactic order, but in execution flow order.
3. If statements create two flows to be followed.
4. Static variables (the *TI* interpreter program) are removed. (not stupid PE).

*The actual pieces of code in the compiler are from mix.*

For each *TI* instruction, rather than doing what the *TI* interpreter, does, the *compiler* does what *mix* does: i.e., it produces specialized versions of *mix,* which generate code.


23


24

## The ACID test: PE for C

*Extremely Tricky. Some problems:*

1. Functions are not idempotent!!!

```
int main() {
  int x;
  push(e1);
  push(e2);
  x = pop() + pop();
  push(x);
}
```

```
int sp=0, stack[100];
void push(int v) {
  stack[++sp] = v;
}
int pop(void) {
  return stack[sp--];
}
```

Solution:

a) Depth first Expansion..specialize the function for each expansion! Function call must be specalized before the statements following the call.

b) Memoization works, as long as side-effects are "memoized" as well.

c) Unfold memo-ized calls.

Yields:

```
int main() {
  stack[1] = e1 ;
  stack[2] = e2 ;
  x = stack[1] + stack[2];
  stack[1] = x;
}
```

*Where would this particular specialization be paying off nicely?*


24

25

## ACID test continued

*Arrays!* How do you statically evaluate expressions involving arrays, e.g:

```
(p[++i]) + (q[j++])
```

Two types of analysis:

1. Can it be determined what p[i] and q[j] are pointing to statically?
2. Can it be determined what *values* p[i] and q[j] have statically?

**Example:**

Assume in the previous case, for acceses to the array `stack` we can determine the subscript statically. Now, we have:

```
stack[4] = .      →      stack_4 = .
..
stack[5] = ..     →      stack_5 = ..
..
stack[6] = ..     →      stack_6 = ..
..
```

*Why is this a good idea?*
*Can we do even better than this? when?*
*Under what circumstances can this be done?*
*What would be effect of this be on a stack based interpreter (stack for the expression evaluation, with bounded depth).*


25


26

## Specializing Structs.

What if some parts of a struct are static, and others dynamic? *Split it!* E.g., consider the following table, for counting # words in input from a fixed dictionary:

```
struct key {char *word, int count;} keytab[NKEYS];
```

words are static, but count is dynamic, so we want to treat them separately; (remember, ↑ static ↑ specialization!)

***(LOOK THIS UP in K&R BOOK)***


26

27

## Taming Pointers

A pointer is *static* if it points just to objects with a location known at specialization time; *dynamic* otherwise.

```
int x, y,z,*p;
P = &x;
*p = 2;
y = *p;
*p = dynExp1;
z = p;
```

```
int a[10], *p;
p = &a[dynExp2];
```

*What's a static pointer above? what's a dynamic pointer? How can these two cases be specialized?*

Tricky problems with scoping and hygiene:

```
int *glob;
int trompFun(int i){
   int j;
   ....
   ....
   glob = &j;
   /* computations with glob; */
   return(some exp);
}
```

```
main() {
  int i,j;
  ......
  i = *glob;
  ......
  .....
}
```


27


28

## Heap allocations...

When the *entire* allocation of a recursive datastructure can be determined statically, then why do any runtime allocation? (e.g., Futamura 1 with a source..)

Example 1: One simple allocation in a function, heap allocation can become stack allocation. *Why is this better?*

Example 2: Fixed number of allocations, all pointers are known to be static. *By analogy to arrays!*


28

29

## Some Recent Applications

- IRISA Group: Selective broadcast (implicit invocation) replaced by explicit invocation. Efficiency gains.
- Publish/Subscribe (using pattern match), pattern match is optimized into simple string processing.
- In Sun RPC, invocations of highly parametrized marshalling routines are first specialized, and then inlined.
- Layered libraries (such as Don Batory's Genvoca components)--layers removed by successive inlining.
- Java Virtual Machine:
  1. Stack access (expression evaluation stack replaced by RAM
  2. Virtual function calls de-virtualized by class. hierarchy analysis; some associated run-time typechecking eliminated.
  3. Methods inlined.
  4. Some array bounds checking eliminated.


29