Partial Evaluation

Given program

$P(a_1, \ldots, a_n)$

and a subset of P's inputs

$x_1, \ldots, x_k$

produce new program

$P'(a_{k+1}, \ldots, a_n)$

such that

$P'(x_{k+1}, \ldots, x_n) = P(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$

Goal: P′ runs much faster than P

- useful if **static** inputs much less frequently varying than **dynamic** inputs

Method:

- evaluate code of P depending only on $x_1, \ldots, x_k$ at partial evaluation-time
  - aggressively inline, loop-unroll, "constant" propagate & fold, specialize
- leave other **residual** code behind as P′

Examples

Usually, an interpreter program + data to be interpreted

Program	Static Data	Dynamic Data
circuit simulator	circuit	initial conditions
cache simulator	cache configuration	address trace
graphics renderer	scene, perspective, light sources	pixel location
string matcher	string pattern	input string
file read & write kernel routines	inode# and file properties	file contents

E.g.

```
circuit-sim(circuit, inputs)
```

$PE(\texttt{circuit-sim}, \texttt{circuit}) \Rightarrow \texttt{circuit-sim}_{\texttt{circuit}}$

$\texttt{circuit-sim}_{\texttt{circuit}}(\texttt{inputs}) =$

```
  circuit-sim(circuit, inputs)
```

Partial evaluation of interpreters

Partial-evaluate an interpreter w.r.t. a program

Want program P s.t.

`"P"(input) = output`

Easy implementation path: interpretation

`Interp(P, input) = output`

Note that `Interp` applied to `P` for many different inputs ⇒
use PE to make interpretation happen faster

`PE(Interp, P)` ⇒ $\texttt{Interp}_{\texttt{P}}$

$\texttt{Interp}_{\texttt{P}}\texttt{(input) = output}$

What is $\texttt{Interp}_{\texttt{P}}$?

(1st Futamura projection)

PE of PE

Previous:

`PE(Interp, P)` ⇒ $\texttt{Interp}_{\texttt{P}}$

$\texttt{Interp}_{\texttt{P}}\texttt{(input) = output}$

`PE` may be applied to `Interp` many times, for different `P`'s ⇒
use PE to make partial evaluation happen faster

`PE(PE, Interp)` ⇒ $\texttt{PE}_{\texttt{Interp}}$

$\texttt{PE}_{\texttt{Interp}}\texttt{(P)}$ ⇒ $\texttt{Interp}_{\texttt{P}}$

$\texttt{Interp}_{\texttt{P}}\texttt{(input) = output}$

Self-application of PE!

What is $\texttt{PE}_{\texttt{Interp}}$? What role does `Interp` play?

(2nd Futamura projection)

Again

Previous:

`PE(PE, Interp)` $\Rightarrow$ $\texttt{PE}_{\texttt{Interp}}$

$\texttt{PE}_{\texttt{Interp}}\texttt{(P)} \Rightarrow \texttt{Interp}_{\texttt{P}}$

$\texttt{Interp}_{\texttt{P}}\texttt{(input)} = \texttt{output}$

`PE` may be applied to `PE` many times, for different `Interp`'s $\Rightarrow$ use PE to make partial evaluation happen faster

`PE(PE, PE)` $\Rightarrow$ $\texttt{PE}_{\texttt{PE}}$

$\texttt{PE}_{\texttt{PE}}\texttt{(Interp)} \Rightarrow \texttt{PE}_{\texttt{Interp}}$

$\texttt{PE}_{\texttt{Interp}}\texttt{(P)} \Rightarrow \texttt{Interp}_{\texttt{P}}$

$\texttt{Interp}_{\texttt{P}}\texttt{(input)} = \texttt{output}$

What is $\texttt{PE}_{\texttt{PE}}$?

(3rd Futamura projection; why no 4th Futamura projection?)

On-line PE

On-line PE is much like a souped-up compiler

- maintain whole data structures as constants (use Lisp/Scheme as base language)
- constant fold expressions involving constants
- residualize code involving dynamic non-constant or residualized operands

For calls, a choice:

- unfold (inline-expand)
- residualize
  - multiple specialized versions if **polyvariant** PE

Termination issues

Example:

```
proc exp(x, n)   -- x^n
  if n = 1 then 1
  else x * exp(x, n-1);
```

PE(`exp`, `n`=3) ⇒ `x * x * x`

PE(`exp`, `x`=2) ⇒ ???

Can't unfold each recursive `exp` call in second example
But don't want to residualize recursion in first example

Solution:

- unfold, unless recursive inside a dynamic `if`

PE still might not terminate

- e.g. if input program contains infinite loop somewhere

Off-line PE

Split PE into two phases:

- binding-time analysis
- specialization

BTA annotates each variable & expression
with **static** or **dynamic**,
given initial annotation on program arguments

Specialization evaluates static computations and residualizes dynamic ones

How to avoid non-termination?

Pragmatics

PE doesn't always work well

Program to be PE'd must have good **binding-time separation**:
static computations must be separated from dynamic compuations

Output of PE in same language as program being PE'd
(composed of extracts from program)

- makes it hard for PE to generate low-level code

PE's today handle functional languages, I/O side-effects, higher-order functions, partially-static data structures

- some work on PE for imperative languages

Idea: apply PE at run-time, using run-time data

- PE time must be repaid with execution savings $\Rightarrow$
  good for infrequently-changing, frequently-accessed data