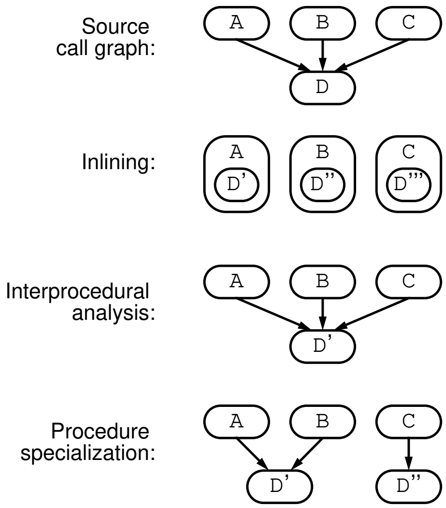
Procedure specialization

A.k.a. procedure cloning, customization

Halfway between inlining and interprocedural analysis



Abstract process

Given set of call sites of procedure P

e.g. $\{c_1, c_2, c_3, c_4, c_5\}$

Partition into equivalence classes of "similar" call sites

e.g. $\{\{c_1, c_2\}, \{c_3, c_4\}, \{c_5\}\}$

Copy P for each class, change calls accordingly

Do interprocedural analysis on changed call graph

Versus inlining:
- \+ less code explosion
- \+ works in presence of recursion

Versus interprocedural analysis:
- \+ better optimization

Question: how to do partitioning?

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

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

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'

Goal: P' runs much faster than P
- faster if lots of computation depends only on static inputs, not dynamic ones
- useful if static inputs often reused for different dynamic inputs

Examples

Usually, an interpreter program + static 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.

$\texttt{circuit-sim(circuit, inputs)}$

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

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

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

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

(1st Futamura projection)

PE of PE

Previous:
$\texttt{PE(Interp, P)} \Rightarrow \texttt{Interp}_\texttt{P}$
$\texttt{Interp}_\texttt{P}\texttt{(input) = output}$

`PE` may be applied to `Interp` many times, for different `P`’s ⇒
use PE to make partial evaluation happen faster
$\texttt{PE(PE, Interp)} \Rightarrow \texttt{PE}_\texttt{Interp}$
$\texttt{PE}_\texttt{Interp}\texttt{(P)} \Rightarrow \texttt{Interp}_\texttt{P}$
$\texttt{Interp}_\texttt{P}\texttt{(input) = output}$

Self-application of PE!

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

(2nd Futamura projection)

Again

Previous:
$\texttt{PE(PE, Interp)} \Rightarrow \texttt{PE}_\texttt{Interp}$
$\texttt{PE}_\texttt{Interp}\texttt{(P)} \Rightarrow \texttt{Interp}_\texttt{P}$
$\texttt{Interp}_\texttt{P}\texttt{(input) = output}$

`PE` may be applied to `PE` many times, for different `Interp`’s ⇒
use PE to make partial evaluation happen faster
$\texttt{PE(PE, PE)} \Rightarrow \texttt{PE}_\texttt{PE}$
$\texttt{PE}_\texttt{PE}\texttt{(Interp)} \Rightarrow \texttt{PE}_\texttt{Interp}$
$\texttt{PE}_\texttt{Interp}\texttt{(P)} \Rightarrow \texttt{Interp}_\texttt{P}$
$\texttt{Interp}_\texttt{P}\texttt{(input) = output}$

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

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

On-line PE

E.g. Fuse [Weise, Ruf, *et al.*]

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

- do constant propagation, but where constants can be whole data structres
  - support partially-static data structures, too
- constant-fold expressions involving constants, including accesses of elements of constant aggregates
- residualize code involving dynamic (non-constant) or residualized operands

For calls, a choice:

- unfold (inline-expand)
- residualize
  - multiple specialized versions if **polyvariant** PE
  - otherwise **monovariant** PE = simple interprocedural analysis

Termination issues

Example:

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

PE(`exp`, `n`=3) $\Rightarrow$ `x * x * x`

PE(`exp`, `x`=2) $\Rightarrow$???

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. PE(`exp`, `n`=-1)

Off-line PE

E.g. Similix [Danvy *et al.*], Schism [Consel *et al.*]

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

Versus on-line PE:

+ can reuse BTA for multiple specializations
+ for some reason, better suited for self-application
– lower quality BTA,
since can't reason about static inputs' values

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
computations

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;
less 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
- if know what data will be fixed at compile-time,
do BTA at compile-time, only specialize at run-time

[E.g., Auslander *et al.* 96, Consel & Noel 96, Leone & Lee 96]