Partial Evaluation

These notes are primarily based on a tutorial by Consel and Danvy.

Partial evaluation is a program transformation that specializes a program with respect to part of its input.

Idea is to remove layers of interpretation, and to evaluate things at compile time rather than waiting until runtime.

Applications of Partial Evaluation (PE)

- compilation and automatic compiler generation
- string and pattern matching
- graphics
- scientific (numerical) applications
- circuit simulation
- hard real-time systems

PE Example

Example: formatted printed function (shown in Scheme in the paper)

```
void fprintf (stream format arg1...)
   for offset = 1 to number of token in format
       switch (get-token(format offset))
            ... integer
            ... string
            ... newline
            ...
```

Given a use:

```
fprintf(f, "%d is not %s\n", x, s);
```

The compiler may generate a specialized version of `fprintf`:

```
void fprintf.1 (stream arg1 arg2)
   print integer arg1
   print string "is not %s\n"
   print string arg2
   print newline
```

Space vs. Time

PE is based on the common space/time trade-off

```
more code space          ----          more compact code
less time                                more time
```

Sometimes you can win in both directions, but often PE can lead to large code blowups.

Advantages of Partial Evaluation

The arguments `f, x, and s` will be filled in at runtime, but the overhead of interpreting `fprintf` itself is eliminated.

Through specialization, the compiler may:

- propagate constants
- unroll the loop
- eliminate branches statement
- string code size (of the given routine)

In general, many standard optimizations are enable when some previously unknown variables become constants.

Structure of Source Programs

A program specializes easily when it processes the static part of the input independently of the dynamic:

```
(lambda (x y z)                (lambda (x y z)
  (+ (+ x y) z))                  (+ x (+ y z)))
```

Specializing with respect to x=2 and y=3:

```
(lambda (z)                (lambda (z)
  (+ 5 z))                  (+ 2 (+ 3 z)))
```

There is some work in restructing programs to improve binding times.

Specific Systems for Functional Languages

Fuse by Daniel Weise

- applied to practical problems like circuit simulation
- addresses code blowup problem
- based on an intermediate language of graphs for better sharing

Similix by Bondorf and Schism by Consel

- offline polyvariant
- self-applicable (at lesat Similix)
- higher order function and partially static values

Systems for Imperative Languages

What languages?

- Pascal [Meyer]
- Real-Time systems [Nirkhe and Pugh]
- C subset with binding time annotations [Anderson]

Problem with PE on imperative programs: lack of referential transparency. Need to be careful of evaluation order.

- Duplication of side effects
- Aliasing

Termination

PE can loop in two ways:

- unfolding infinitely many function calls
- creating infinitely many specializations

Arbitrary limits can be used to avoid the problem, but most use something less ad hoc.

- Mix - insert annotations to indicate unfolding, later automated analysis
- Schism - filters associated with each function to determine whether it should be unfolded
- Fuse - using an online strategy, keeps a cache of program points – specialization stops on a cache hit (+ arbitrary bound)

Pattern Matching

PE is useful, for example, in pattern matching.

Problem: Does a string occur in some text?

Linear time algorithms due to Knuth, Morris & Pratt and by Boyer and Moore can be seen as partial evaluations on naive n^2 algorithm.

Idea: after a partial match, we know that a piece of the dynamic input string is identical to a prefix of the static pattern. Now match the pattern against a shifted copy of itself, determined at specialization time.

Certain shifts are eliminated by matching the pattern against itself, so at any given point, the next shift can be precomputed based on the pattern.

KM&P - left-to-right traversal of string B&M - right-to-left

Generalizing the Partial Evaluator

```
(lambda (s d)
  (case d
  [(1) (+ s d)]
  [(2) (- s d)]
  [else d]
```

Specialize for s = 10.

Naive:

```
(lambda (d)
  (case d
  [(1) (+ 10 d)]
  [(2) (- 10 d)]
  [else d]
```

Better:

```
 (lambda (d)
   (case d
     [(1) 11]
     [(2) 8]
     [else d]))
```

PE and OS

OS trend: restructing monolithic kernels into independent servers running on top of a minimal/micro kernel.

Usual advantage of modularity, plus high level languages used, but there is a performance cost.

- Synthesis kernel by Pu – efficiency regained
- Consel et al – automatic programming from specifications
- Tic C language by Kaashok – language support for the Exo kernel.