Online Partial Evaluation / Function Specialization

Program Transformation 2005–2006

Eelco Visser

Institute of Information & Computing Sciences
Utrecht University
The Netherlands

March 2, 2006

Online Partial Evaluation

Goal

- Inputs: program, inputs for some arguments
- Output: program specialized to these inputs, optimized as much as possible

Simplification

- Input: program with constant expressions
- Output: program specialized to constant expressions, optimized as much as possible

Online vs Offline

- Online: decision to specialize based on program + inputs
- Offline: decision to specialize based on program + declaration of static inputs

Outline

- Refine constant propagation strategy to online specializer
- Specialize 0: constant propagation
- Specialize 1: function unfolding
- Specialize 2: unfold only static calls
- Specialize 3: memoize call unfolding
- Specialize 4: specialization of function definitions
- Specialize 5: reduction of specialized functions

Specialize 0: Constant Propagation

```
pe = PropConst <+ pe-assign <+ pe-declare
     <+ pe-let <+ pe-if <+ pe-while <+ pe-for
     <+ all(pe); try(EvalBinOp <+ EvalRelOp <+ EvalString)

pe-assign =
  |[ x := <pe => e> ]|
  ; if <is-value> e
    then rules( PropConst.x :  |[ x ]| -> |[ e ]| )
    else rules( PropConst.x :- |[ x ]| ) end

pe-declare =
  ? |[ var x ta ]|
  ;  rules( PropConst+x :- |[ x ]| )

pe-declare =
  |[ var x ta := <pe => e> ]|
  ; if <is-value> e
    then rules( PropConst+x :  |[ x ]| -> |[ e ]| )
    else rules( PropConst+x :- |[ x ]| ) end

pe-let =
  |[ let <*id> in <*id> end ]|
  ; {| PropConst : all(pe) |}
```

Specialize 0: Constant Propagation (control flow)

```
pe-if =
  |[ if <pe> then <id> ]|
  ; (EvalIf <+ (|[ if <id> then <pe> ]| /PropConst\ id))

pe-if =
  |[ if <pe> then <id> else <id> ]|
  ; (EvalIf; pe
     <+ (|[ if <id> then <pe> else <id> ]|
               /PropConst\ |[ if <id> then <id> else <pe> ]|))

pe-while =
  |[ while <id> do <id> ]|
  ; (|[ while <pe> do <id> ]|; EvalWhile
     <+ /PropConst\* |[ while <pe> do <pe> ]|)
```

Specialize 1: Function Unfolding

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in printint(fact(10))
end
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else n * fact(n - 1)
 in printint(3628800)
end
```

Specialize 1: Function Unfolding
Replace call and substitute arguments

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in printint(fact(10))
end
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in printint(
      if 10 < 1 then 1 else (10 * fact(10 - 1))
    )
end
```

Specialize 1: Function Unfolding — Substitution

```
pe = PropConst <+ {| PropConst : UnfoldCall; pe |} <+ ...

pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ e ]|
        where <zip(SubstArg)> (x*, a*) => d*
    )

SubstArg =
  ?(FArg|[ x ta ]|, e)
  ; rules( PropConst : |[ x ]| -> |[ e ]| )
```

Specialize 1: Function Unfolding — Substitution

```
pe = PropConst <+ {| PropConst : UnfoldCall; pe |} <+ ...

pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ e ]|
        where <zip(SubstArg)> (x*, a*) => d*
    )

SubstArg =
  ?(FArg|[ x ta ]|, e)
  ; rules( PropConst : |[ x ]| -> |[ e ]| )
```

Substitution may lead to duplication of computations and side effects.

Specialize 1: Function Unfolding — Let Binding

```
pe = PropConst <+ UnfoldCall; pe <+ ...

pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ let d* in e end ]|
        where <zip(BindArg)> (x*, a*) => d*
    )

BindArg :
  (FArg|[ x ta ]|, e) -> |[ var x ta := e ]|
```

Specialize 1: Function Unfolding — Let Binding

```
pe = PropConst <+ UnfoldCall; pe <+ ...

pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ let d* in e end ]|
        where <zip(BindArg)> (x*, a*) => d*
    )

BindArg :
  (FArg|[ x ta ]|, e) -> |[ var x ta := e ]|
```

Binding expressions to variable and subsequent constant propagation have same effect as substitution, but is safe.

Specialize 1: Function Unfolding — Replace call by body

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in printint(fact(10))
end
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in printint(
      let var n := 10
       in if n < 1 then 1 else (n * fact(n - 1))
      end)
end
```

Specialize 1: Function Unfolding — Constant propagation

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in printint(
      let var n := 10
       in if n < 1 then 1 else (n * fact(n - 1))
      end)
end
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in printint(
      let var n := 10
       in 10 * fact(9)
      end)
end
```

Specialize 1: Function Unfolding — Constant Fold Let

```
EvalLet :
  |[ let d* in i end ]| -> |[ i ]|

pe-let =
  |[ let <*id> in <*id> end ]|
  ; {| PropConst : all(pe) |}
  ; try(EvalLet)
```

If let body reduces to a constant value, the bindings are dead.

Specialize 1: Function Unfolding — Cleaning up

```
pe-let =
  |[ let <*id> in <*id> end ]|
  ; {| PropConst, UnfoldCall : all(pe) |}
  ; |[ let <*filter(not(DeadBinding))> in <*id> end ]|
  ; try(EvalLet)

DeadBinding =
  ?|[ var x ta := x ]|

DeadBinding =
  ?|[ var x ta := i ]|

EvalLet :
  |[ let d* in i end ]| -> |[ i ]|
```

Remove useless variable bindings

```
let ...
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

```
 in printint(
      let var x : int := readint()
       in x * let var x : int :=
                     let var x : int := x * 1
                      in x * x end
                in x * x end
              end)
end
```

Specialize 1

```
let function square(x : int) : int =
      x * x

    function mod(x : int, y : int) : int =
      x - (x / y) * y

    function even(x : int) : int =
      mod(x, 2) = 0

    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))

 in printint(power(6, readint())); print("\n")

end
```

Problem: Specialize 1 does not terminate for many programs

Specialize 2: Unfold only calls with all static arguments

```
pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ let d* in e end ]|
        where <map(is-value)> a*
            ; <zip(BindArg)> (x*, a*) => d*
    )
```

```
let
   function fibonacci(n:int):int =
     if (n >= 2) then
       fibonacci(n-1) + fibonacci(n-2)
     else if (n = 1) then
       1
     else 0
in printint(fibonacci(30))
end
```

Problem: re-evaluation of same static calls

Specialize 3: memoize call unfoldings

```
unfold-call :
  |[ f(a*) ]| -> |[ e ]|
  where <RetrieveUnfolding> |[ f(a*) ]| => |[ e ]|
     <+ <UnfoldCall; pe> |[ f(a*) ]| => |[ e ]|
        ; rules(
            RetrieveUnfolding.f : |[ f(a*) ]| -> |[ e ]|
          )

pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ let d* in e end ]|
        where <map(is-value)> a*
            ; <zip(BindArg)> (x*, a*) => d*

      RetrieveUnfolding+f
    )
```

memoization works

fib	Specialize2	Specialize3
15	0m1.656s	0m0.219s
17	0m3.955s	0m0.227s
20	0m14.906s	0m0.232s

```
let function square(x : int) : int =
      x * x

    function mod(x : int, y : int) : int =
      x - (x / y) * y

    function even(x : int) : int =
      mod(x, 2) = 0

    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))

 in printint(power(readint(), 5)); print("\n")

end
```

Problem: Specialize 3 does not specialize partially static calls

Specialize4: specialization of function definitions

```
pe = ... <+ all(pe); try(... <+ SpecializeCall)

pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ let d* in e end ]|
        where <map(is-value)> a*
            ; <zip(BindArg)> (x*, a*) => d*

      RetrieveUnfolding+f

      Specialization+f

      SpecializeCall :
        |[ f(a1*) ]| -> |[ g(a2*) ]|
        where // next slide
    )
```

Specialize4: specialize function to static arguments

```
pe-declare = ?|[ function f(x*) ta = e ]|;
rules(
  SpecializeCall :
    |[ f(a1*) ]| -> |[ g(a2*) ]|
    where <split-static-dynamic-args> (x*, a1*) => (d*, (x2*, a2*))
        ; new => g; !e => e2
        ; rules(
            Specialization.f :+
              |[ function f(x*) ta = e' ]| ->
              |[ function g(x2*) ta = let d* in e2 end ]|
          )
)
split-static-dynamic-args =
  zip; partition(BindArgValue); (not([]), unzip)

BindArgValue :
  (FArg|[ x ta ]|, e) -> |[ var x ta := e ]| where <is-value> e
```

Specialize4: specialize function to static arguments

```
pe-declare = ?|[ function f(x*) ta = e ]|;
rules(
  SpecializeCall :
    |[ f(a1*) ]| -> |[ g(a2*) ]|
    where <split-static-dynamic-args> (x*, a1*) => (d*, (x2*, a2*))
        ; new => g; !e => e2
        ; rules(
            Specialization.f :+
              |[ function f(x*) ta = e' ]| ->
              |[ function g(x2*) ta = let d* in e2 end ]|
          )
)
split-static-dynamic-args =
  zip; partition(BindArgValue); (not([]), unzip)

BindArgValue :
  (FArg|[ x ta ]|, e) -> |[ var x ta := e ]| where <is-value> e
```

separate static from dynamic arguments

Specialize4: specialize function to static arguments

```
pe-declare = ?|[ function f(x*) ta = e ]|;
rules(
  SpecializeCall :
    |[ f(a1*) ]| -> |[ g(a2*) ]|
    where <split-static-dynamic-args> (x*, a1*) => (d*, (x2*, a2*))
        ; new => g; !e => e2
        ; rules(
            Specialization.f :+
              |[ function f(x*) ta = e' ]| ->
              |[ function g(x2*) ta = let d* in e2 end ]|
          )
)
split-static-dynamic-args =
  zip; partition(BindArgValue); (not([]), unzip)

BindArgValue :
  (FArg|[ x ta ]|, e) -> |[ var x ta := e ]| where <is-value> e
```

separate static from dynamic arguments

`R :+ l -> r` — dynamic rule with *multiple* right-hand sides

Specialize4: replace functions with their specialization

```
pe-let =
  |[ let <*id> in <*id> end ]|
  ; {| PropConst, UnfoldCall, RetrieveUnfolding, Specialization :
       all(pe)
       ; |[ let <*filter(|[ <fd*:mapconcat(bagof-Specialization)> ]|
                          <+ not(DeadBinding))>
              in <*id> end ]|
     |}
  ; try(EvalLet)
```

Specialize4: replace functions with their specialization

```
pe-let =
  |[ let <*id> in <*id> end ]|
  ; {| PropConst, UnfoldCall, RetrieveUnfolding, Specialization :
       all(pe)
       ; |[ let <*filter(|[ <fd*:mapconcat(bagof-Specialization)> ]|
                          <+ not(DeadBinding))>
              in <*id> end ]|
     |}
  ; try(EvalLet)
```

bagof-R: *all* rewritings of R

```
let function square(x : int) : int =
      x * x
    function mod(x : int, y : int) : int =
      x - (x / y) * y
    function even(x : int) : int =
      mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

```
let function a_0(x : int) : int =
      let var n : int := 5
       in if n= 0 then 1
          else if even(n) then square(power(x, n / 2))
               else x * power(x, n - 1)
      end
 in printint(a_0(readint()))
end
```

```
let function square(x : int) : int =
      x * x
    function mod(x : int, y : int) : int =
      x - (x / y) * y
    function even(x : int) : int =
      mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

```
let function a_0(x : int) : int =
      let var n : int := 5
       in if n= 0 then 1
          else if even(n) then square(power(x, n / 2))
               else x * power(x, n - 1)
      end
 in printint(a_0(readint()))
end
```

Problem: body of specialized function is not transformed

Specialize5: reduce body of specialized function

```
pe-declare = ?|[ function f(x*) ta = e ]|;
rules(
  ...
  SpecializeCall :
    |[ f(a1*) ]| -> |[ g(a2*) ]|
    where <split-static-dynamic-args>
              (x*, a1*) => (d*, (x2*, a2*))
        ; new => g
        ; <pe>|[ let d* in e end ]| => e2
        ; rules(
            Specialization.f :+
              |[ function f(x*) ta = e' ]| ->
              |[ function g(x2*) ta = e2 ]|
          )
)
```

Specialize5: reduce body of specialized function

```
pe-declare = ?|[ function f(x*) ta = e ]|;
rules(
  ...
  SpecializeCall :
    |[ f(a1*) ]| -> |[ g(a2*) ]|
    where <split-static-dynamic-args>
              (x*, a1*) => (d*, (x2*, a2*))
        ; new => g
        ; <pe>|[ let d* in e end ]| => e2
        ; rules(
            Specialization.f :+
              |[ function f(x*) ta = e' ]| ->
              |[ function g(x2*) ta = e2 ]|
          )
)
```

transform instantiated body before declaring specialization

```
let function square(x : int) : int =
      x * x
    function mod(x : int, y : int) : int =
      x - (x / y) * y
    function even(x : int) : int =
      mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

```
let function a_0(x : int) : int = x * d_0(x)
    function d_0(x : int) : int = square(e_0(x))
    function e_0(x : int) : int = square(g_0(x))
    function g_0(x : int) : int = x * h_0(x)
    function h_0(x : int) : int = 1
 in printint(a_0(readint()));
    print("\n")
end
```

```
let function square(x : int) : int =
      x * x
    function mod(x : int, y : int) : int =
      x - (x / y) * y
    function even(x : int) : int =
      mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

```
let function a_0(x : int) : int = x * d_0(x)
    function d_0(x : int) : int = square(e_0(x))
    function e_0(x : int) : int = square(g_0(x))
    function g_0(x : int) : int = x * h_0(x)
    function h_0(x : int) : int = 1
 in printint(a_0(readint()));
    print("\n")
end
```

Problem: lots of ‘trivial’ functions generated

Summary and Future Work

Summary

- Partial evaluation extends constant propagation to functions
- Specialization of function to constant arguments
- Find balance between function unfolding and specialization
- *Extended dynamic rules* to hold multiple specializations

Future work: improvements to partial evaluator

- Try to unfold as many function calls as possible
- Doing too much will lead to non-termination
- How to control unfolding?
- Memoization to catch calls to specializations in progress
- Offline partial evaluation: use binding time analysis to decide which calls to unfold and which to specialize