Program Specialization

Ralf Lämmel
What’s program specialization?

Program specialization is a semantics-based, powerful, and general optimization technique. It has applications in model-driven development, domain-specific language engineering, and generic programming.

Program specialization also goes by the name “partial evaluation”. The kind of partial evaluation we do here is called “online partial evaluation”.

© Ralf Lämmel, 2009-2012 unless noted otherwise
Simple ideas for program specialization for a simple functional language

Allow undefined variables during evaluation.
Preserve syntactical phrases if they cannot be evaluated completely.

Consider a function for exponentiation:
exp x n = if n==0 then 1 else x * exp x (n-1)

Assume n = 3. We would like to partially evaluate exp as follows:
exp' x = x * x * {x * 1}

BTW, trivial optimization applicable
A simple functional programming language: syntax and interpreter
Syntax of a simple functional language

```haskell
type Prog = ([FDef], Expr)

type FDef = (String, ([String], Expr))

data Val
  = IVal { getInt :: Int }
  | BVal { getBool :: Bool }

data Expr
  = Const { getVal :: Val }
  | Var String
  | Apply String [Expr]
  | Prim Op [Expr]
  | If Expr Expr Expr

data Op = Equal | Add | Sub | Mul
```

What does it mean to partially evaluate the given expression in an environment that specifies \( y = \) and with no binding for \( x \)? The result is the residual code if \( x > 0 \) then \( 10/x \) else \( 0 \)? In this case: both branches of the conditional must be partially evaluated: if \( y \) is to be eliminated? This is not the normal evaluation rule for conditionals? As a result: partial evaluation may diverge where regular evaluation would terminate—unless extra effort is made? Also: great care must be taken in the presence of side effects? Consider again the expression given earlier for a partial environment such that \( x = \) and with no binding for \( y \)? Partial evaluation is supposed to result in the residual code if \( 0 > y \) then \( \text{error} \) else \( y \)? Note that the division, by zero error must not be raised during partial evaluation: but it must be delayed so that it is executed at the right point in the residual code—if it is ever exercised? Function calls and recursive definitions also cause complications: which are discussed in the tutorial?

The style of partial evaluator developed here is called an online partial evaluator: because it makes decisions about specialization as it goes: based on whatever variables are in the environment at a given point during evaluation [5]? An offline partial evaluator performs a static analysis of the program to decide which variables will be considered known and which will be considered unknown? It has been claimed that offline partial evaluation is simpler than online partial evaluation [(—)]? It can be more difficult to ensure termination in an online setting? In this tutorial: we avoid issues of termination: so this complexity does not arise [9]? Given this restriction: we believe that the online style is simpler to describe: because the partial evaluator can be derived easily from an existing interpreter?
A sample program

```haskell
exp(x, n) = if n == 0 then 1 else x * exp(x, n - 1)
```

```
exp = ( "exp", (["x","n"],
    If (Prim Equal [Var "n", Const (IVal 0)])
    (Const (IVal 1))
    (Prim Mul
        [Var "x",
         Apply "exp" [Var "x", Prim Sub [Var "n", Const (IVal 1)]]]))
)
```

```
> eval ([exp], Apply "exp" [Const (IVal 2), Const (IVal 3)])
8
```
The interpreter (Part I/II)

```haskell

\textbf{type} \hspace{0.5em} \texttt{Env} = [(\texttt{String}, \texttt{Val})]

eval :: \texttt{Prog} \to \texttt{Val}
eval (fdefs, \texttt{main}) = eval' \texttt{main} []

\textbf{where}

eval' :: \texttt{Expr} \to \texttt{Env} \to \texttt{Val}
eval' (\texttt{Const v}) \texttt{env} = v

eval' (\texttt{Var s}) \texttt{env} =
  \textbf{case} \texttt{lookup s} \texttt{env} \textbf{of}
  \hspace{1em} \texttt{Just v} \to v
  \hspace{1em} \texttt{Nothing} \to \texttt{error "undefined variable"}

eval' (\texttt{Prim op es}) \texttt{env} =
  \textbf{let} \texttt{rs} = [eval' e \texttt{env} | e \leftarrow \texttt{es}] \textbf{in}
  \texttt{prim op rs}
```

Figure 2: An evaluator for Fig. 1
The interpreter (Part I/II)

eval’ (If e0 e1 e2) env =
  if getBool (eval’ e0 env)
    then eval’ e1 env
    else eval’ e2 env

eval’ (Apply f es) env =
  eval’ body env’
  where
    (ss, body) = fromJust (lookup f fdefs)
    env’ = zip ss [ eval’ e env | e <- es ]

prim Equal [IVal i1, IVal i2] = BVal (i1 == i2)
prim Add [IVal i1, IVal i2] = IVal (i1 + i2)
prim Sub [IVal i1, IVal i2] = IVal (i1 - i2)
prim Mul [IVal i1, IVal i2] = IVal (i1 * i2)
Naive partial evaluation by inlining
Modification of the eval function

—— Interpreter
eval :: Prog -> Val

—— Partial evaluator
peval :: Prog -> Expr

—— Interpreter
type Env = [(String, Val)]

—— Naive, inlining-oriented partial evaluator
type Env = [(String, Expr)]
The naive, partial evaluator (Part I/II)

\[
\text{type} \ Env = \ [(\text{String}, \ Expr)]
\]

peval :: Prog \rightarrow Expr
peval (fdefs, main) = peval’ main []

where

peval’ :: Expr \rightarrow Env \rightarrow Expr

peval’ (Const v) env = Const v

peval’ (Var s) env =
  \text{case} \ \text{lookup} \ s \ \text{env} \ \text{of}
  \text{Just} \ e \rightarrow e
  \text{Nothing} \rightarrow \text{Var} \ s \ -- \ \text{return code for variable}

peval’ (Prim op es) env =
  \text{let} \ rs = [ \ \text{peval’} \ e \ \text{env} | e \leftarrow es ] \ \text{in}
  \text{if} \ \text{all} \ \text{isVal} \ rs
  \text{then} \ \text{Const} \ (\text{prim} \ \text{op} \ (\text{map} \ \text{getVal} \ rs))
  \text{else} \ \text{Prim} \ \text{op} \ rs \ -- \ \text{return code for primitive}
The naive, partial evaluator (Part II/II)

peval' (If e0 e1 e2) env =
  let r0 = peval' e0 env in
  if isVal r0
    then if getBool (getVal r0)
      then peval' e1 env
      else peval' e2 env
    else (If r0
             (peval' e1 env)
             (peval' e2 env)) —— return code for if

peval' (Apply f es) env =
peval' body env'
where
  (ss, body) = fromJust (lookup f fdefs)
  env' = zip ss [ peval' e env | e <- es ]
What's the problem?

The treatment of conditionals and function applications is naive. For example, partial evaluation of a function application may diverge when compared to regular evaluation. Thus:

```haskell
> peval ([exp], Apply "exp" [Const (IVal 2), Var "n"])
```

--- Result shown in regular Haskell notation for clarity

```haskell
if n == 0
  then 1
else 2 * (if n-1 == 0
  then 1
  else 2 * (if (n-1)-1 == 0 ...))
```

In this example, the function `exp` is applied to a specific base 2, but the exponent remains a variable `n`. Inlining diverges because the recursive case of `exp` is continuously exercised for different argument expressions for the exponent.
Proper program specialization
Proper program specialization

Proper treatment of recursive functions requires from us to synthesize residual programs instead of just residual expressions based on naive inlining. Hence, our more advanced partial evaluator uses the following type:

\[
peval :: \text{Prog} \rightarrow \text{Prog}
\]

Also, we return to the original definition of Env, which binds variables to values rather than expressions.

\[
\text{Env} = [(\text{String}, \text{Val})]
\]

\[
\text{Env} = [(\text{String}, \text{Expr})]
\]
Proper program specialization

The idea is that the incoming function definitions and the main expression are *specialized* such that the resulting main expression only refers to specialized function definitions. The same original function definition may be specialized several times depending on the encountered, statically known argument values. For instance:

```haskell
> peval ([exp], Apply "exp" [Var "x", Const (IVal 3)])

exp’a x = x * exp’b x
exp’b x = x * exp’c x
exp’c x = x * exp’d x
exp’d x = 1
```

Main expression **Apply “exp’a” [Var “x”]**
Proper program specialization

— Naive inlining transformation
peval' :: Expr -> Env -> Expr

— Proper program specialization
peval' :: Expr -> Env -> State [FDef] Expr

Monadic style with state monad for collecting specialized functions
The proper program specializer (Part I/III)

```haskell

type Env = [(String, Val)]

peval :: Prog -> Prog
peval (fdefs, main) = swap (runState (peval' main [[[]]]) []

where

peval' :: Expr -> Env -> State [FDef] Expr

peval' (Const v) env = return (Const v)

peval' (Var s) env =
  case lookup s env of
    Just v -> return (Const v)
    Nothing -> return (Var s)

peval' (Prim op es) env = do
  rs <- mapM (flip peval' env) es
  if all isVal rs
    then return (Const (prim op (map getVal rs)))
    else return (Prim op rs)
```

Thus, the original definition of \textit{exp} was specialized such that the argument position for the statically known base is eliminated. Please note that the specialized function is recursive.

The partial evaluator needs to aggregate specialized functions along with recursion into expressions. To this end, we use the state monad in the type of the expression-level function \textit{peval'}. Thus:

\begin{itemize}
  \item Naive inlining transformation
  \item Proper program specialization
\end{itemize}

The cases for all constructs but function application can be adopted from the simpler partial evaluator— except that we need to convert to monadic style, which is a simple, systematic program transformation in itself [6, 1]. See Fig. 4 for the result. That is, recursive calls to \textit{peval'} are not used directly in
The proper program specializer (Part II/III)

peval’ (If e0 e1 e2) env = do
  r0 <- peval’ e0 env
  if isVal r0 then
    if getBool (getVal r0)
      then peval’ e1 env
      else peval’ e2 env
  else do
    r1 <- peval’ e1 env
    r2 <- peval’ e2 env
    return (If r0 r1 r2)
The proper program specializer (Part III/III)

peval' (Apply s es) env = do
  -- Look up function.
  let (ss, body) = fromJust (lookup s fdefs)

  -- Partially evaluate arguments.
  rs <- mapM (flip peval' env) es

  -- Determine static and dynamic arguments.
  let z = zip ss rs
  let sas = [ (s, getVal r) | (s, r) <- z, isVal r ]
  let das = [ (s,v) | (s,v) <- z, not (isVal v) ]

  if null das then
    -- Inline completely static applications.
    peval' body sas
  else do
    -- Otherwise these applications make their contexts dynamic.
    -- Fabricate name from static variables.
    let s' = s ++ show (hashString (show sas))

    -- Specialize each "name" just once.
    fdefs <- get
    when (isNothing (lookup s' fdefs)) (do
      -- Create placeholder for memoization.
      put (fdefs ++ [{s', undefined}])

      -- Partially evaluate function body.
      e' <- peval' body sas

      -- Replace placeholder by actual definition.
      modify (update (const (map fst das, e')) s'))

    return (Apply s' (map snd das))

Figure 5: Partial evaluation of function application
peval' (Apply s es) env = do

  -- Look up function.
  let (ss, body) = fromJust (lookup s fdefs)

  -- Partially evaluate arguments.
  rs <- mapM (flip peval' env) es

  -- Determine static and dynamic arguments.
  let z = zip ss rs
  let sas = [ (s, getVal r) | (s, r) <- z, isVal r ]
  let das = [ (s,v) | (s,v) <- z, not (isVal v) ]

  if null das
    then
      -- Inline completely static applications.
      peval' body sas
    else do
      -- Fabricate name from static variables.
      let s' = s ++ show (hashString (show sas))
      -- Specialize each znamez just once.
      fdefs <- get when isNothing (lookup s' fdefs))
      -- Create placeholder for memoization.
      put (fdefs ++ ['s', undefined])
      -- Partially evaluate function body.
      e' <- peval' body sas
      -- Replace placeholder by actual definition.
      modify update const map fst das, e')) s'))
      -- Return application of specialized function.
      return (Apply s' (map snd das))
if null das then

    -- Inline completely static applications.
    peval' body sas

    -- Otherwise these applications make their contexts dynamic.
else do
— Otherwise these applications make their contexts dynamic.

else do

— Fabricate name from static variables.
let s’ = s ++ show (hashString (show sas))

— Specialize each "name" just once.
fdefs <- get when (isNothing (lookup s’ fdefs)) (do

— Create placeholder for memoization.
put (fdefs ++ [(s’, undefined)])

— Partially evaluate function body.
e’ <- peval’ body sas

— Replace placeholder by actual definition.
modify (update (const (map fst das, e’)) s’))

— Return application of specialized function.
return (Apply s’ (map snd das))
The solved termination problem

Let us apply the more advanced, partial evaluator to the diverging example that we faced at the end of the previous section. Function specialization carefully tracks argument lists for which specialization is under way or has been completed.

```haskell
> peval ([exp], Apply "exp" [Const (IVal 2), Var "n"])

["exp’", ["n"],
  If (Prim Equal [Var "n", Const (IVal 0)])
    (Const (IVal 1))
    (Prim Mul
      [Const (IVal 2),
        Apply "exp’" [Prim Sub [Var "n", Const (IVal 1)]]]),
      Apply "exp’" [Var "n"]
)
```

© Ralf Lämmel, 2009-2012 unless noted otherwise
A more interesting application of program specialization
Representation of finite state machines

```haskell
type State = Int
type Label = Char
type Transitions = [(State, [(Label, State)])]

type Accept = [State]

transitions = [(1, [('a', 2)]),
               (2, [('b', 1)])]

accept = [2]
```

Figure 6: Types for state machines and example

```haskell
run :: State -> Accept -> Transitions -> [Label] -> Bool
run current accept transitions [] = current 'elem' accept
run current accept transitions ul:lsx = case
    lookup l ufromJust ulookup current transitions xx
    of
        Nothing -> False
        Just next -> run next accept transitions ls

Figure 7: Haskell interpreter for state machines

```haskell
run1 ls =
    if
        null ls
        then
            False
        else if
            head ls == 'a'
            then
                run2 utail lsx
            else
                False

run2 ls =
    if
        null ls
        then
            True
        else if
            head ls == 'b'
            then
                run1 utail lsx
            else
                False

Figure 8: Desired output from partial evaluation

The desired result from partially evaluating the state machine interpreter on the state machine in Fig. 6 is given in Fig. 8. The accept states and the transition table are no longer present as data structures—thereby promising an aggressive optimization. However, when our partial evaluator is applied to the program in Fig. 6 (appropriately encoded in our simple language), specialization fails to eliminate the data structures for accept states and transition table. The problem is this expression:

```haskell
lookup current transitions
```
Interpreter for finite state machines

run :: State -> Accept -> Transitions -> [Label] -> Bool
run current accept transitions [] = current ‘elem’ accept
run current accept transitions (l:ls) =
  case lookup l (fromJust (lookup current transitions)) of
    Nothing -> False
    Just next -> run next accept transitions ls

The desired result from partially evaluating the state machine interpreter on the state machine in Fig. 6 is given in Fig. 8. The accept states and the transition table are no longer present as data structures—thereby promising an aggressive optimization. However, when our partial evaluator is applied to the program in Fig. 6 (appropriately encoded in our simple language), specialization fails to eliminate the data structures for accept states and transition table. The problem is this expression:

lookup current transitions
Specialization of the interpreter for a specific finite state machine

\begin{verbatim}
run1 ls = if null ls
    then False
    else if head ls == 'a'
        then run2 (tail ls)
        else False

run2 ls = if null ls
    then True
    else if head ls == 'b'
        then run1 (tail ls)
        else False
\end{verbatim}

The original interpreter is not ready to be specialized like this. We need to rewrite it a bit for “binding time improvements”; see the tutorial paper for details.
• **Summary**: Program specialization
  ✦ Denotational semantics gives interpreters.
  ✦ Interpreters can be turned into optimizers.
  ✦ Program specialization goes beyond basic inlining.

• **Reading**: “Tutorial on Online Partial Evaluation”
  by William R. Cook and Ralf Lämmel