Unidad 3. Optimización

Las técnicas de optimización buscan mejorar la eficiencia de los programas que se materializa en menores tiempos de ejecución y en menor consumo de recursos. Una técnica popular para la optimización de programas es la especialización o evaluación parcial de programas.

"La Evaluación Parcial (EP) es una técnica de transformación automática de programas que persigue, entre otros objetivos, la optimización de programas con respecto a ciertos datos de entrada; de ahí que también haya recibido el nombre de especialización de programas" (Pascual Julián, 2003)

En la siguiente figura se ilustra un evaluador parcial.

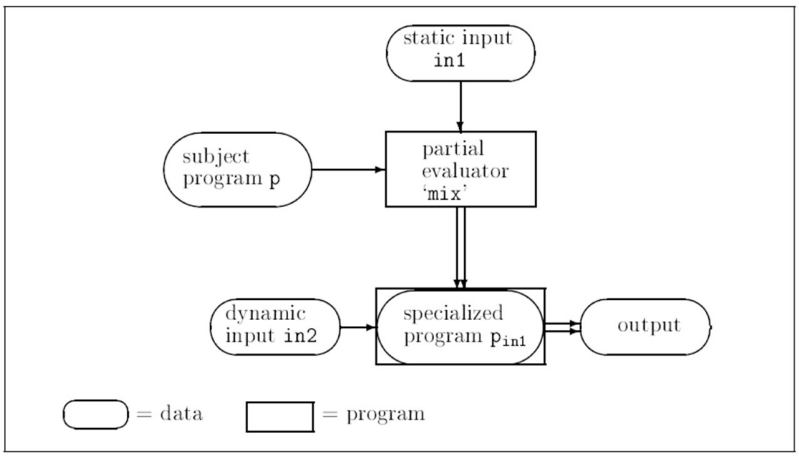

Figure 1.1: A partial evaluator.

Si se considera a P como un programa general que requiere dos argumentos, esto es P(*a1,a2*), y a *a1* como un argumento, entonces la ejecución de un programa especializado P_{a1} junto con el dato a2 será mas eficiente que la ejecución del programa P con sus dos argumentos, formalmente:

P_{a1} (a2) >e P(a1,a2), donde >e significa mayor eficiencia

Especializar un programa no es una tarea fácil, se trata de hacer una transformación de código mediante la ejecución parcial de instrucciones que dependen de los datos conocidos. Se le considera de las técnicas de optimización más recientes.

Los datos conocidos antes de tiempo de ejecución se llaman datos estáticos y los datos que se conocen hasta tiempo de ejecución se llaman datos dinámicos.

En la figura, dada la entrada de un programa p y una entrada estática in1, un evaluador parcial produce un programa especializado p_{in1} el cual al recibir el dato dinámico in2, en tiempo de ejecución, produce una salida output, igual a la salida que habría producido ejecutar p con in1, y con in2, pero en mejor tiempo de ejecución.

Esto es, la evaluación parcial busca generar programas especializados mas eficientes, para casos específicos.

Enseguida se plantean algunos ejemplos:

Dada la forma general de la función potencia

```
pot b exp = if (exp==0) then 1
                else  b * (pot b  (exp -1))
```

Considere que se conoce el valor del exponente, entonces generar una versión especializada para el exponente 3. La evaluación parcial será de la siguiente manera

Fase de despliegue: unfolding

```
Sustituyendo:  pot b 3 = if (3==0) then 1
                               else  b * (pot b  (3 -1))
Reduciendo:     pot b 3 =  b * (pot b  2)
```

Sustituyendo: pot b 2 = if (2==0) then 1
else b * (pot b (2 -1))
Reduciendo: pot b 2 = b * (pot b 1)

Sustituyendo: pot b 1 = if (1==0) then 1
else b * (pot b (1 -1))
Reduciendo: pot b 1 = b * (pot b 0)

Sustituyendo: pot b 0 = if (0==0) then 1
else b * (pot b (0 - 1))
Reduciendo: pot b 0 = 1

Copiando las reglas generadas tenemos:

pot b 3 = b * (pot b 2)
pot b 2 = b * (pot b 1)
pot b 1 = b * (pot b 0)
pot b 0 = 1

Renombramiento de las llamadas a función que tienen datos conocidos

pot b 3 será pot_3 b
pot b 2 será pot_2 b
pot b 1 será pot_1 b
pot b 0 será pot_0 b

Sustituyendo los nuevos nombres en las reglas generadas tenemos:

$pot_3\ b = b * (pot_2\ b)$
$pot_2\ b = b * (pot_1\ b)$
$pot_1\ b = b * (pot_0\ b)$
$pot_0\ b = 1$

Aplicando la fase de post unfolding queda:

$pot_3\ b = b * (pot_2\ b)$
$b * (pot_1\ b)$
$b * (pot_0\ b)$
1
$pot_3\ b = b * (b*(\ b*(\ 1)\ )\ )$

La versión especializada para pot b exp, cuando sabemos que el exponente es 3, entonces es:

$pot_3\ b = b * (b*(\ b*(\ 1)\ )\ )$

De aquí resulta evidente que **pot_3 b** es mas eficiente que **pot b exp** para el caso de que el exponente sea 3.

EJEMPLO 2, ESPECIALIZACIÓN DE LISTAS

Considere el caso general de la función **append x y** para concatenar dos lista x y ye:

append [] y = y
append (x:xs) y = x:(append xs y)

Consideremos ahora que conocemos el primer argumento cuyo valor es: [1,2,3] genere una función especializada $append_{123}$ para append [1,2,3] y

Fase de despliegue: unfolding

append (1:[2,3]) y = 1: (append [2,3] y)
append (2:[3]) y = 2: (append [3] y)
append (3:[ ]) y = 3: (append [] y)
append [] y = y

Renombramiento de las llamadas a función que tienen datos conocidos

append (1:[2,3]) será $append_{123}$
append (2:[3]) será $append_{23}$
append (3:[ ]) será $append_{3}$
append [] será $append_{\phi}$

Sustituyendo los nuevos nombres en las reglas generadas tenemos:

$append_{123}$ y = 1: ($append_{23}$ y)
$append_{23}$ y = 2: ($append_{3}$ y)
$append_{3}$ y = 3: ($append_{\phi}$ y)
$append_{\phi}$ y = y

Aplicando la fase de post unfolding queda:

$append_{123}$ y = 1: ($append_{23}$ y)
2: ($append_{3}$ y)
3: ($append_{\phi}$ y)
y

$append_{123}$ y = 1: (2: (3: (y)))

La versión especializada para **append x y**, cuando sabemos que la lista x es [1,2,3] queda:

$append_{123}$ y = 1: (2: (3: (y)))

De aquí resulta evidente que **$append_{123}$ y = 1: (2: (3: (y)))** es mas eficiente que **append x y** para el caso de que la lista inicial sea [1,2,3].

[1] Pascual Julián Iranzo, Novática: Revista de la Asociación de Técnicos de Informática, ISSN 0211-2124, Nº. 163, 2003 (Ejemplar dedicado a: Conocimiento abierto. Open knowledge), págs. 40-46.