



# Parseo y Generación de Código

9 de noviembre de 2017

**Análisis estático**  
**Optimización**

Licenciatura en Informática con Orientación en Desarrollo de Software  
Universidad Nacional de Quilmes

# Análisis estático

# Análisis estático

**Objetivo.** Determinar propiedades comunes a todas las posibles ejecuciones de un programa, sin ejecutarlo.

## Ejemplo.

```
fun f(b : Bool) : Int {  
  y := 3  
  if (b) {  
    y := 7  
  } else {  
    y := 2 * y  
    y := y + 1  
  }  
  x := 14 / y      •  
  return x  
}
```

En cualquier ejecución del programa,  $y$  vale 7 en la línea •.

Esto puede servir para hacer optimizaciones. Por ejemplo, reemplazar “ $x := 14 / y$ ” por “ $x := 2$ ”.

# Análisis estático

Prácticamente todos los problemas de análisis estático son **indecidibles**.

**Definición.** Un conjunto de programas  $X$  se dice un *conjunto de índices* si para cualquier par de programas  $p_1$ ,  $p_2$  equivalentes (es decir, si  $p_1$  y  $p_2$  computan la misma función), se tiene que:

$$p_1 \in X \iff p_2 \in X$$

**Teorema de Rice.** Sea  $X$  un conjunto de índices no trivial, es decir,  $X$  no es vacío ni es el conjunto de todos los programas. Entonces el conjunto  $X$  no es computable, es decir, no existe un programa  $p$  que, dado otro programa  $q$ , determine si  $q \in X$  o bien  $q \notin X$ .

**Nota.** Este teorema sirve para justificar que la mayor parte de los problemas de análisis estático son indecidibles. Enunciarlo precisamente y demostrarlo excede el alcance de esta materia.

# Análisis estático

- **Solución exacta.** Un problema de análisis estático (p.ej. “¿el programa hace alguna división por cero?”) tiene solución exacta si el resultado del análisis se corresponde siempre con el comportamiento del programa en tiempo de ejecución:

Resultado del análisis	Realidad
“sí”	sí
“no”	no

Dado que los problemas de análisis estático en general son indecidibles, **no es posible resolverlos de manera exacta.**

Casi todos los análisis dan como respuesta una **aproximación.**

# Análisis estático

Hay dos maneras típicas de aproximar la solución a un problema:

- ▶ **Análisis “may”**. El resultado del análisis indica que **puede** cumplirse una condición en alguna ejecución del programa.

Resultado del análisis	Realidad	
“puede que sí”	sí / no	es posible que se cumpla
“no ”	no	no es posible que se cumpla

- ▶ **Análisis “must”**. El resultado del análisis indica que **debe** cumplirse una condición en todas las ejecuciones del programa.

Resultado del análisis	Realidad	
“sí”	sí	es seguro que se cumple
“puede que no”	sí / no	no es seguro que se cumple

# Grafos de flujo de control

Hay muchas técnicas y variantes de análisis estático. Nos vamos a concentrar en el **análisis de flujo de datos intraprocedural**.

Para hacer análisis de flujo de datos, típicamente se construye primero el **grafo de flujo de control**.

Un **bloque básico** es una secuencia de instrucciones tales que:

- ▶ No hay ninguna instrucción de salto en el programa que llegue a una instrucción en el medio del bloque (sólo pueden llegar al inicio de la secuencia).
- ▶ El bloque no contiene ninguna instrucción de salto, o alternativamente contiene una única instrucción de salto al final.

En otras palabras, un bloque básico es una secuencia de instrucciones en las que el flujo de control es lineal: siempre entra al principio y sale al final, sin detenerse y sin posibilidad de saltar excepto al final.

# Grafos de flujo de control

El **grafo de flujo** de un programa es un grafo dirigido:

- ▶ Los nodos son bloques básicos.
- ▶ Hay una arista del bloque  $B$  al bloque  $B'$  si el control puede pasar del final del bloque  $B$  al comienzo del bloque  $B'$ .

**Ejercicio.** Escribir el siguiente programa en código de tres direcciones y calcular su grafo de flujo de control.

```
-- n es un parámetro
p := 1
i := 0
while i < n {
    if i mod 2 == 0 {
        i := i + 1
        p := p * i
    } else {
        i := i + 1
    }
}
```



# Análisis de flujo de datos

El análisis de flujo de datos es una técnica general que sirve para asociar información (es decir, un conjunto de datos) a cada punto del programa. *P. ej.: “conjunto de variables que valen cero”.*

Se basa en analizar estáticamente todos los posibles caminos en el grafo de flujo de control.

Cada bloque básico  $B$  tiene asociados cuatro conjuntos de datos:

- ▶  $in(B)$  – información disponible al inicio del bloque  $B$ .
- ▶  $out(B)$  – información disponible al final del bloque  $B$ .
- ▶  $gen(B)$  – información que se genera al ejecutar el bloque  $B$ .
- ▶  $kill(B)$  – información que se destruye al ejecutar el bloque  $B$ .

En general  $gen(B)$  y  $kill(B)$  son datos, mientras que  $in(B)$  y  $out(B)$  son incógnitas.

# Análisis de flujo de datos

Cada problema de análisis de flujo de datos determina ecuaciones entre los conjuntos *in*, *out*, *gen* y *kill* de los bloques básicos.

Resolviendo los sistemas de ecuaciones asociados se pueden calcular valores para las incógnitas *in* y *out*.

En líneas generales se pueden identificar dos clases de análisis:

- ▶ **Forward (“hacia adelante”)**. El conjunto  $out(B)$  se calcula a partir del conjunto  $in(B)$ .
- ▶ **Backward (“hacia atrás”)**. El conjunto  $in(B)$  se calcula a partir del conjunto  $out(B)$ .

# Análisis de flujo de datos

## Ecuaciones de flujo de datos.

En general siempre que se tengan dos bloques  $B_1$ ,  $B_2$  del grafo de flujo:

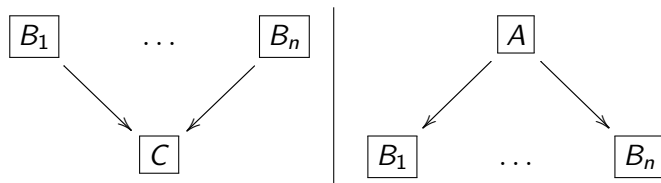


se debe agregar la ecuación  $out(B_1) = in(B_2)$ .

# Análisis de flujo de datos

## Ecuaciones de flujo de datos.

Más en general, los puntos en los que el flujo de control confluye o diverge (*join points*) tienen asociadas ecuaciones:



La forma de las ecuaciones depende del tipo de análisis:

	forward	backward
<b>may</b>	$in(C) = \bigcup_{i=1}^n out(B_i)$	$out(A) = \bigcup_{i=1}^n in(B_i)$
<b>must</b>	$in(C) = \bigcap_{i=1}^n out(B_i)$	$out(A) = \bigcap_{i=1}^n in(B_i)$

# Análisis de flujo de datos

## Ecuaciones de flujo de datos.

Además, para cada bloque  $B$  se incorpora la siguiente ecuación:

- ▶ Para el análisis forward:

$$out(B) = gen(B) \cup (in(B) \setminus kill(B))$$

- ▶ Para el análisis backward:

$$in(B) = gen(B) \cup (out(B) \setminus kill(B))$$

Así obtenemos un **sistema de ecuaciones de flujo de datos**.

# Análisis de flujo de datos

A continuación veremos tres ejemplos de análisis de flujo de datos:

1. Definiciones de alcance.
2. Variables vivas.
3. Expresiones disponibles.

## Ejemplo: *definiciones de alcance*

Una **definición de alcance** (*reaching definition*) para una instrucción  $\mathcal{I}_1$  que lee una variable  $x$  es otra instrucción  $\mathcal{I}_0$  que escribe la variable  $x$ , de tal manera que, en alguna ejecución del programa, al ejecutar  $\mathcal{I}_1$  se tiene que  $\mathcal{I}_0$  es la última instrucción que le dio valor a  $x$ .

Por ejemplo:

```
1  i := 0
2  p := 1
3  loop_start:
4  jumpIf≥ i n loop_end
5  i := i + 1
6  p := p * i
7  jump loop_start
8  loop_end:
```

- ▶ ¿Cuáles son las definiciones de alcance para  $i$  en la línea 6?
- ▶ ¿Cuáles son las definiciones de alcance para  $p$  en la línea 6?

## Ejemplo: *definiciones de alcance*

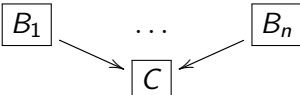
Se puede aproximar con un análisis **may forward**.

El conjunto de datos es un conjunto de pares  $(x, k)$  donde  $x$  es una variable y  $k$  es un número de línea que es una *posible definición* de  $x$ .

Ecuaciones de flujo de datos:

- Por ser hacia adelante:

$$out(B) = gen(B) \cup (in(B) \setminus kill(B))$$

- Por ser de tipo *may*: 

$$in(C) = out(B_1) \cup \dots \cup out(B_n)$$

- Una escritura  $x := e$  en la línea  $k$  genera el dato  $(x, k)$  y destruye todos los datos de la forma  $(x, k')$ .



## Ejemplo: *definiciones de alcance*

**Ejercicio.** Dado el siguiente programa:

```
1  i := 0
2  p := 1
3  loop_start:
4  jumpIf $\geq$  i n loop_end
5  i := i + 1
6  p := p * i
7  jump loop_start
8  loop_end:
```

- ▶ Construir el grafo de flujo.
- ▶ Calcular las definiciones de alcance al principio y al final de cada bloque básico, resolviendo las ecuaciones de flujo correspondientes.

## Ejemplo: *definiciones de alcance*

### **Aplicaciones de las definiciones de alcance.**

Una vez que se calcularon las definiciones de alcance, esta información se puede utilizar para hacer optimizaciones o emitir *warnings*. Por ejemplo:

- ▶ Si se tiene una instrucción  $x := 100 \text{ div } z$  tal que entre las definiciones de alcance para  $z$  hay una instrucción  $z := 0$ , se puede emitir un *warning* indicando que *posiblemente* haya una división por cero.
- ▶ Si se tiene una instrucción  $x := 2 * z$  y todas las instrucciones de alcance para  $z$  son de la forma  $z := 10$ , se puede reemplazar la instrucción original por  $x := 20$ .

## Ejemplo: *definiciones de alcance*

**Ejemplo.** En el siguiente programa:

```
1  if (siempreDevuelveTrue()) {  
2      x := 2  
3  } else {  
4      x := 0  
5  }  
6  y := 10 / x
```

Supongamos que `siempreDevuelveTrue()` es una función que siempre devuelve `True`.

- ▶ El conjunto de definiciones de alcance para `x` en la línea 6 es el conjunto  $\{(x, 2)\}$ .
- ▶ El análisis reporta el conjunto  $\{(x, 2), (x, 4)\}$ , que es una **sobreaproximación**, y el compilador podría dar un falso *warning* de división por cero.

En general `siempreDevuelveTrue()` podría ser una función muy compleja, y no hay manera general de diseñar un analizador que identifique que siempre devuelve `True`.

## Ejemplo: *variables vivas*

Una variable  $x$  está **viva** en un punto del programa  $p$  si en alguna posible ejecución del programa se utiliza el valor de  $x$  vigente en  $ep$  (antes de que su valor se sobrescriba). Por ejemplo:

```
1  x := 10
2  y := 10 * x
3  z := y + 1
4  x := 1
5  z := z + x
```

¿Cuáles son las variables vivas en cada punto del programa?

## Ejemplo: *variables vivas*

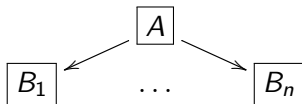
Se puede aproximar con un análisis **may backward**.

El conjunto de datos es un conjunto de variables que están *posiblemente* vivas.

- Por ser hacia atrás:

$$in(B) = gen(B) \cup (out(B) \setminus kill(B))$$

- Por ser de tipo *may*:



$$out(A) = in(B_1) \cup \dots \cup in(B_n)$$

- La escritura de una variable  $x$  la destruye:  
 $kill(x := y + z) = \{x\}$ .
- La lectura de una variable  $x$  la genera:  
 $gen(x := y + z) = \{y, z\}$ .

## Ejemplo: *variables vivas*

**Ejercicio.** Dado el siguiente programa:

```
1  a := 5
2  c := 1
3  .loop_start:
4  jumpIf $\leq$  c a .loop_end
5  a := 5
6  c := c + c
7  jump .loop_start
8  .loop_end:
9  a := 2 * a
10 c := 0
```

- ▶ Construir el grafo de flujo.
- ▶ Calcular las variables vivas al principio y al final de cada bloque básico, resolviendo las ecuaciones de flujo correspondientes.

## Ejemplo: *variables vivas*

### **Aplicaciones de las variables vivas.**

Una vez que se calcularon las variables vivas, esta información se puede utilizar para hacer optimizaciones o emitir *warnings*. Por ejemplo:

- ▶ Si al inicio del primer bloque de un procedimiento hay una variable local  $x$  viva, quiere decir que  $x$  es una variable que *posiblemente* nunca está declarada. El compilador puede emitir un *warning* en ese caso.
- ▶ Si hay dos variables  $x$  e  $y$  que nunca están simultáneamente vivas (es decir, cuando  $x$  está viva,  $y$  no está viva), se puede utilizar el mismo registro para almacenarlas. Esto permite reducir la utilización de registros.
- ▶ Una instrucción de asignación  $x := E$  se puede eliminar si  $x$  no está viva. (*Dead store elimination*).

## Ejemplo: *variables vivas*

**Ejemplo.** En el siguiente programa:

```
1  if (siempreDevuelveTrue()) {  
2      x := 1  
3  }  
4  y := x
```

- ▶ Al principio del programa la variable  $x$  no está viva, porque siempre se le asigna un valor antes de la línea 4.
- ▶ Sin embargo, el análisis reporta que  $x$  está viva. El compilador podría dar un falso *warning* indicando que  $x$  está potencialmente no declarada.



## Ejemplo: *expresiones disponibles*

En un punto del programa  $p$  una expresión  $E$  está **disponible** en la variable  $x$  si en todas las posibles ejecuciones del programa que llegan al punto  $p$  hay una instrucción que calcula  $x := E$ , de tal modo que entre dicha asignación y el punto  $p$  nunca se modifica el valor de la variable  $x$  ni el valor de las variables involucradas en la expresión  $E$ .

Por ejemplo:

```
1  a := 10
2  b := a + a
3  c := b + 1
4  a := a + a
5  b := a + a
```

¿Cuáles son las expresiones disponibles en cada punto del programa?

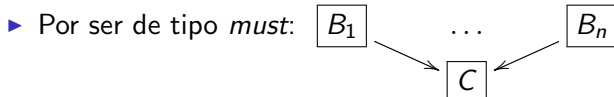
## Ejemplo: *expresiones disponibles*

Se puede aproximar con un análisis **must forward**.

El conjunto de datos asociado a un punto del programa es un conjunto de pares  $(x, E)$  que representan el hecho de que la expresión  $E$  se encuentra disponible en la variable  $x$ .

- ▶ Por ser hacia adelante:

$$out(B) = gen(B) \cup (in(B) \setminus kill(B))$$



$$in(C) = out(B_1) \cap \dots \cap out(B_n)$$

- ▶ Una asignación  $x := E$  genera  $(x, E)$  si  $x \notin E$ .
- ▶ Una asignación  $x := E$  destruye  $(y, E')$  si  $(x \in y) \vee (x \in E')$ .

## Ejemplo: *expresiones disponibles*

**Ejercicio.** Dado el siguiente programa:

```
1  i := 1
2  d := 2 * i
3  .loop_start:
4  jumpIf> a n .loop_end
5      i := i + 1
6      d := 2 * i
7  jump .loop_start
8  .loop_end:
9  r := 2 * i
```

- ▶ Construir el grafo de flujo.
- ▶ Calcular las expresiones disponibles al principio y al final de cada bloque básico, resolviendo las ecuaciones de flujo correspondientes.

## Ejemplo: *expresiones disponibles*

### **Aplicaciones de las expresiones disponibles.**

Una vez que se calcularon las expresiones disponibles, esta información se puede utilizar, típicamente, para hacer optimizaciones.

Por ejemplo, si la expresión  $E$  está disponible en la variable  $x$ , la asignación  $y := E$  se puede reemplazar por una asignación  $y := x$ , ahorrando ese cómputo.

# Solución algorítmica para ecuaciones de flujo

El siguiente método se puede utilizar para resolver ecuaciones de flujo de datos, suponiendo que se trata de un análisis **forward**, ya sea **may** (definiendo  $\diamond := \cup$ ) o **must** (definiendo  $\diamond := \cap$ ):

*Entrada:* Un grafo de flujo de control  $G$ ,  
conjuntos  $gen(B)$  y  $kill(B)$  para cada bloque  $B$  de  $G$ .  
*Salida:* Conjuntos  $in(B)$  y  $out(B)$  para cada bloque  $B$  de  $G$ .

Poner  $in(B) := \emptyset$  y  $out(B) := \emptyset$  para todo bloque  $B$ .

```
while hay algún cambio
  foreach bloque  $B$ 
    Sea  $\{A_1, \dots, A_n\}$  el conjunto de los bloques
      tales que hay una arista  $A_i \rightarrow B$  en  $G$ .
     $in(B) := out(A_1) \diamond \dots \diamond out(A_n)$ 
     $out(B) := gen(B) \cup (in(B) \setminus kill(B))$ 
  end
end
```

# Solución algorítmica para ecuaciones de flujo

Si el análisis es **backward**, el código es análogo, usando la variante de las ecuaciones que propagan la información desde  $out(B)$  hacia  $in(B)$ .

Para que este método termine, deben cumplirse ciertas condiciones. En los casos que estudiamos, basta observar que:

- ▶ En cada iteración del algoritmo, los conjuntos  $in(B)$  y  $out(B)$  nunca se achican.
- ▶ Dado un programa fijo, hay una cantidad finita de conjuntos posibles que pueden decorar los nodos  $in(B)$  y  $out(B)$ .

Esto vale para los tres análisis vistos (definiciones de alcance, variables vivas, expresiones disponibles).

## Comentario: generalización del análisis de flujo de datos

Un **reticulado** es un conjunto  $X$  dotado de una relación de orden  $\sqsubseteq$  y de operaciones  $\sqcup$  y  $\sqcap$  que cumplen:

$$x \sqsubseteq x \sqcup y$$

$$x \sqcap y \sqsubseteq x$$

$$y \sqsubseteq x \sqcup y$$

$$x \sqcap y \sqsubseteq y$$

$$(x \sqsubseteq z \wedge y \sqsubseteq z) \implies x \sqcup y \sqsubseteq z \quad (z \sqsubseteq x \wedge z \sqsubseteq y) \implies z \sqsubseteq x \sqcap y$$

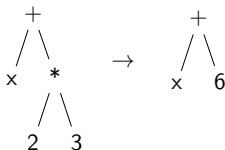
El análisis de flujo de datos se puede generalizar para que cada nodo del grafo de flujo tenga asociado un elemento en un reticulado.

Otras técnicas de optimización



## Plegado de constantes

Las subexpresiones que involucran solamente constantes se pueden **plegar**. Esta transformación puede hacerse a nivel del AST:



O puede hacerse a nivel del código intermedio, usando definiciones de alcance para saber cuáles son los valores posibles que puede tomar una variable:

```
t1 := x
t2 := 2
t3 := 3
t2 := t2 * t3
t1 := t1 + t2
```

plegado de constantes →

```
t1 := x
t2 := 2
t3 := 3
t2 := 6
t1 := t1 + t2
```

eliminación de  
asignaciones a  
variables muertas →

```
t1 := x
t2 := 6
t1 := t1 + t2
```

## Plegado de constantes

Se pueden utilizar propiedades de las operaciones aritméticas para generalizar el plegado de constantes. Por ejemplo, en la siguiente expresión no se pueden plegar constantes:

$$((x + 1) + 1) + 1$$

pero usando el hecho de que la suma es asociativa, es decir, que  $(x + y) + z = x + (y + z)$ , la expresión de arriba se puede reescribir así:

$$x + (1 + (1 + 1))$$

lo que permite plegar la subexpresión constante:

$$x + 3$$

## Plegado de constantes

El plegado de constantes se puede generalizar para reescribir el programa usando identidades aritméticas:

$$x + 0 \rightarrow x$$

$$x * 0 \rightarrow 0$$

$$x * 1 \rightarrow x$$

**Cuidado.** Hay que ser cuidadoso para aplicar estas transformaciones directamente sobre el AST. Por ejemplo reescribir  $f() * 0$  como 0 no es correcto en un lenguaje en el que  $f()$  puede tener efectos secundarios.

Si el lenguaje tiene efectos secundarios se puede reescribir  $f() * 0$  como `let _ = f() in 0`.

## Plegado de constantes

**Más cuidado.** Hay que tener cuidado con las identidades aritméticas cuando se trabaja con representaciones numéricas típicas en la computadora.

- ▶ Por ejemplo, si  $x$  e  $y$  son enteros de longitud fija con signo, la siguiente reescritura no es válida:

$$(x + y) / 2 \rightarrow x + (y - x) / 2$$

por ejemplo podría ocurrir que  $x$  e  $y$  sean números positivos suficientemente grandes como para que la suma  $(x + y)$  dé *overflow* y se interprete como un número negativo.

- ▶ Otro ejemplo: es bien sabido que la suma de números de punto flotante no es asociativa, es decir, la siguiente igualdad:

$$x + (y + z) = (x + y) + z$$

no es válida en general.

## Plegado de constantes

En lenguajes funcionales típicamente es posible reescribir el programa con otras identidades:

$$\text{map } f \text{ (map } g \text{ xs)} \quad \rightarrow \quad \text{map } (f \text{ . } g) \text{ xs}$$

En este ejemplo, la variante de la derecha suele ser conveniente porque hace un único recorrido sobre la lista.

## Plegado de constantes

El plegado de constantes se puede generalizar para plegar construcciones condicionales y de *pattern matching*:

```
if True {  
  A  
} else {  
  B  
} → A
```

---

```
t1 := 10  
t2 := 11  
jumpIf ≤ t1 t2 .label → t1 := 10  
t2 := 11  
jump .label
```

---

```
head [x1, x2, x3] → x1
```

---

```
case Just "a" of  
  Nothing -> 0 → "a"  
  Just x -> x
```

# Plegado de constantes

El plegado de constantes es una optimización “benévola” en casi todos los sentidos:

- ▶ Reduce el tamaño del código.
- ▶ Mejora el tiempo de ejecución.
- ▶ Reduce el uso de memoria.
- ▶ Puede tener un efecto “cascada”, posibilitando otras optimizaciones:

```
          x * head [0, 1, 2, 3]
→        x * 0
→        0
```

## Compilación para efecto vs. para efecto+valor

Considerar el siguiente programa:

```
fun f(x : Int) {  
    print(x)  
    return 2 * x + 2  
}
```

```
fun main() {  
    f(f(9))  
}
```

Hay dos invocaciones a f:

- ▶ Ambas tienen un efecto y retornan un valor.
- ▶ El valor de la invocación externa nunca se usa.

Algunos compiladores distinguen entre llamados a función **sólo por su efecto** y llamados a función **por su efecto y también por su valor**.



## Compilación para efecto vs. para efecto+valor

El compilador podría generar código para el programa anterior con la siguiente estructura:

```
fun f(x : Int) {  
    print(x)  
    return 2 * x + 2  
}  
  
fun f_solo_efecto(x : Int) {  
    print(x)  
}  
  
fun main() {  
    f_solo_efecto(f(9))  
}
```

- ▶ Esta optimización puede mejorar el tiempo de ejecución y el uso de memoria.
- ▶ Pero puede duplicar el tamaño del código.

# Eliminación de código muerto

El compilador puede eliminar código muerto:

- ▶ Funciones que nunca se invocan.
- ▶ Ramas de un *if* o de un *case* inalcanzables.

Si el código muerto está en el código fuente, se puede reportar un *warning*. El código muerto también puede aparecer como resultado de otras optimizaciones.

## Inlining de funciones

Dada una definición de función:

```
fun f(x1, ..., xn) {  
    cuerpo  
}
```

una invocación  $f(a_1, \dots, a_n)$  se puede reemplazar por una expresión de la forma:

```
let x1 = a1  
    ...  
    xn = an  
in  
    cuerpo
```

Esto se conoce como **inlining**.

## Inlining de funciones

El *inlining* de funciones tiene ventajas y desventajas:

- ▶ Se evita el mecanismo de pasaje de parámetros, creación del registro de activación, apilado y desapilado de registros, y saltos para hacer el `call/return`.
- ▶ El código puede crecer exponencialmente.

El compilador puede aplicar algunas heurísticas para determinar si es conveniente hacer *inlining* de una función:

- ▶ Si la función se utiliza una sola vez en todo el programa, es razonable eliminar su definición y reemplazar su única invocación por su código.
- ▶ Si el cuerpo de la función es muy pequeño, de tal manera que el costo de ejecutar el cuerpo es menor que el trabajo requerido para invocarla, es razonable hacer *inlining*.
- ▶ El compilador debería evitar hacer *inlinings* anidados adentro de *inlinings* de manera indiscriminada.

## Inlining de funciones

El mayor interés de hacer *inlining* de una función es la posibilidad de que esto permita otras optimizaciones posteriores. Por ejemplo:

```
fun f(x : Int, y : Int) {  
    return x * y + 1  
}
```

```
fun g(y : Int) {  
    print(f(f(0, y), y))  
}
```

Si se decide hacer inlining de la invocación interna, se obtiene:

```
fun f(x : Int, y : Int) {  
    return x * y + 1  
}
```

```
fun g(y : Int) {  
    print(f(0 * y + 1, y))  
}
```

## *Inlining* de funciones

Plegando constantes:

```
fun f(x : Int, y : Int) {  
    return x * y + 1  
}
```

```
fun g(y : Int) {  
    print(f(1, y))  
}
```

Haciendo otro *inlining* de f:

```
fun f(x : Int, y : Int) {  
    return x * y + 1  
}
```

```
fun g(y : Int) {  
    print(1 * y + 1)  
}
```

## *Inlining* de funciones

Plegando constantes:

```
fun f(x : Int, y : Int) {  
    return x * y + 1  
}
```

```
fun g(y : Int) {  
    print(y + 1)  
}
```

Eliminando código muerto:

```
fun g(y : Int) {  
    print(y + 1)  
}
```

## Desenrollado de ciclos

Los ciclos se pueden **desenrollar**.

Una forma básica de desenrollado de ciclos es reemplazar un ciclo de  $n$  iteraciones por  $n$  copias del ciclo:

<pre>r := 0 for i = 1 to 5 {   r := r + i }</pre>	$\xrightarrow{\text{desenrollado}}$	<pre>r := 0 r := r + 1 r := r + 2 r := r + 3 r := r + 4 r := r + 5</pre>
---	-------------------------------------	--

Combinado con técnicas de optimización anteriores, por ejemplo plegado de constantes, se pueden obtener mejoras importantes:

```
r := 15
```

La desventaja obvia es que el tamaño del código puede crecer mucho.



# Desenrollado de ciclos

Los ciclos también se pueden desenrollar de manera general, copiando el cuerpo varias veces (por ejemplo, 4 veces):

```
while cond {  
    cuerpo  
}  
  
    desenrollado →  
  
while cond {  
    cuerpo  
    if cond {  
        cuerpo  
        if cond {  
            cuerpo  
            if cond {  
                cuerpo  
            } else break;  
        } else break;  
    } else break;  
}
```

# Desenrollado de ciclos

- ▶ Una ventaja de desenrollar ciclos es que puede aliviar la *predicción de saltos* del procesador.
- ▶ Igual que en los casos anteriores, desenrollar un ciclo puede permitir otras optimizaciones. Por ejemplo, si cada iteración hace  $k := k + 1$ , al plegar constantes y eliminar asignaciones a variables muertas se puede obtener una única asignación  $k := k + 4$ .