Guido A. Macchi
Departamento de Cs. de la Computación
FCEIA – UNR

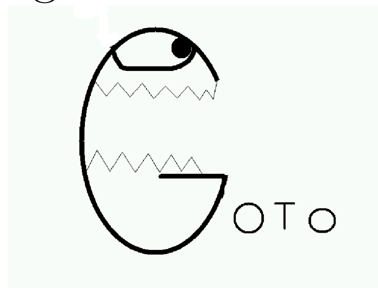
Counterclockwise
Productions
Union

presenta

Counterclockwise
Productions
Union

presenta

Continuaciones:
La Venganza del GOTO



:

¡Llevó cuarenta años filmarla!

¡Llevó cuarenta años filmarla!

Un elenco estelar: Dijkstra, Landin, Strachey, Scott, Reynolds, Steele, Appel y muchos otros.

Una crítica unánime.

Una crítica unánime.

Dijo Fidel ML: *las continuaciones son lo más horrible y antinatural que puede haber.*

Primer Protagonista: el GOTO y las continuaciones

Un `GOTO` es, sencillamente, un salto. Hace que la ejecución de una secuencia continúe en otra instrucción distinta de la que le sigue.

Es, con seguridad, la construcción más criticada en la historia de la informática, luego del famoso artículo de E. Dijkstra *GOTO considered harmful*.

Es, con seguridad, la construcción más criticada en la historia de la informática, luego del famoso artículo de E. Dijkstra *GOTO considered harmful*.

A pesar de todo, se toleran versiones domesticadas, como `call`, `return`, `break`, o `continue`.

En setiembre de 1.964, durante IFIP Working Conference on Formal Language Description, Adriaan van Wijngaarden propuso una mejora en la implementación del mecanismo para invocar funciones, mostrando que se podía descartar un *goto*.

Wijngaarden no tuvo mucho éxito, y ninguno de los asistentes, E. Dijkstra y Doug McIll-roy incluídos, entendió la idea.

De hecho, a diferencia de Wijngaarden, que quiso demostrar que un *goto* era innecesario, Dijkstra dedujo que *todos* los *gotos* eran *indeseables*; nace así el concepto de Programación Estructurada.

Ahora, ¿cuál fue la malograda idea de Wijngaarden?

Imaginemos la siguiente expresión en C:

`h(g(f(1)))`.

Imaginemos la siguiente expresión en C:

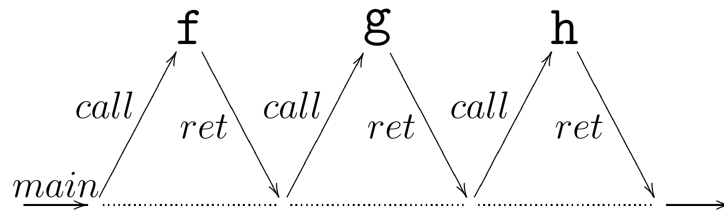
`h(g(f(1)))`.

Típicamente, se implementa como

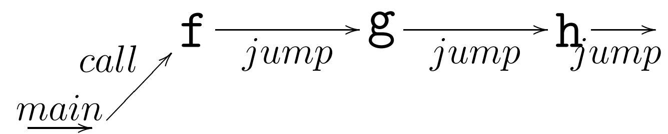
```
push 1       argumento para f
call f       vuelve con ret
add sp, 4    corrige la pila
push rv      valor devuelto, argumento para g
call g
add sp, 4
push rv
call h
add sp, 4
```

donde `rv` es el registro en el que vuelve el valor retornado; las funciones deben terminar con `ret`.

Al ejecutar esto, tendremos la siguiente traza.



Ya que el resultado de una función es el argumento de la siguiente, van Wijngaarden propuso que dicha función no retornara, sino que invocara a la siguiente pasándole su resultado.



Luego de esta transformación, las funciones han dejado de serlo, ya que *no retornan*. ¿Esta transformación es posible siempre? Sí, pasándole a toda función un parámetro extra: la función a la que debe pasarle lo calculado.

Ahora, estas funciones representan *el resto del programa*. Si generalizamos esto a cualquier expresión, el resto del programa es la *continuación* de dicha expresión.

La técnica para convertir expresiones en funciones que pasan sus resultados al resto del programa (sin volver) se conoce como *Continuation Passing Style* o *CPS*.

Ejemplo: CPS para CBV.

Ejemplo: CPS para CBV.

$$\overline{x} \equiv \lambda k.k\ x$$

Ejemplo: *CPS* para *CBV*.

$$\overline{x} \equiv \lambda k.k\ x$$
$$\overline{\lambda x.E} \equiv \lambda k.k(\lambda x.\overline{E})$$

Ejemplo: CPS para CBV.

$$
\begin{aligned}
\overline{x} &\equiv \lambda k.k\ x \\
\overline{\lambda x.E} &\equiv \lambda k.k\ (\lambda x.\overline{E}) \\
\overline{M\ N} &\equiv \lambda k.\overline{M}\ (\lambda f.\overline{N}\ (\lambda a.f\ a\ k))
\end{aligned}
$$

Esto tiene mucha relación con la *Recursión Posterior* o *Tail Recursion*.

Las continuaciones se han usado en compiladores (SMLofNJ), intérpretes (una versión de Python), sistemas operativos (Mach), etc., con buenos resultados.

La ventaja estriba en que *CPS* no necesita *stack* o permite reusarlo, por lo que consume, en gral., menos memoria.

Las continuaciones fueron redescubiertas varias veces, en varios contextos.

- P. Landin (1.965), relacionando ALGOL60 y λ–cálculo..
- A. W. Mazurkiewicz (1.969), trabajando en teoría de autómatas.
- F. L. Morris (1.970), con intérpretes definicionales.
- Ch. Strachey y C. P. Wadsworth (1.970), en semánticas denotacionales.
- J. L. Morris (1.971), en transformaciones de ALGOL.
- J. M. Fisher (1.972), en transformaciones de λ–cálculo.
- S. K. Abdali (1.973), en traducciones de ALGOL a λ–cálculo.

Pero, ¿qué puede ocurrir si un lenguaje *permite* que el programador tenga *acceso* a las continuaciones y manipularlas?

Entre otros, tres lenguajes que permiten esto son

- Scheme,
- Standard ML of New Jersey y
- (de manera restringida) C.

Scheme reifica las continuaciones como funciones, usando

`call-with-current-continuation`

o

`call/cc`;

SMLofNJ lo hace con

`callcc` y `throw`;

C con

`setjmp`, `longjmp` y `setcontext`.

Un ejemplo en C.

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf buf;
void f()
{
   puts("entramos! ");
   longjmp(buf, 1);
   puts("salimos! ");
}
int main()
{
   if(setjmp(buf)!=0) {
      puts("auch! "); return 0;
   }
   puts("uno! "); f(); puts("dos! ");
   return 0;
}
```

Un ejemplo en C.

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf buf;
void f()
{
   puts("entramos! ");
   longjmp(buf, 1);
   puts("salimos! ");
}
int main()
{
   if(setjmp(buf)!=0) {
      puts("auch! "); return 0;
   }
   puts("uno! "); f(); puts("dos! ");
   return 0;
}
```

Aparece `uno! entramos! auch!`.

Un ejemplo con Scheme. La expresión

```
(call/cc(lambda(k) (+ 3 (k 4))))
```

evalúa a 4.

La continuación `k` funciona como un `goto` con valores.

La continuación anterior ha "saltado" hacia adelante (*downward continuation*). Pero, ¿y si devolvemos la continuación como valor (*upward continuation*)? Pues, ¡tenemos la posibilidad de iterar!

(Ejemplo **MUY** oscuro).

(Ejemplo **MUY** oscuro).

```
(define call/cc
   call-with-current-continuation)
```

(Ejemplo **MUY** oscuro).

```
(define call/cc
   call-with-current-continuation)
(define (iter proc)
  (let
    ((vamos (call/cc (lambda(k) k))))
    (begin
      (proc)
      (vamos vamos))))
```

(Ejemplo **MUY** oscuro).

```
(define call/cc
   call-with-current-continuation)
(define (iter proc)
  (let
    ((vamos (call/cc (lambda(k) k))))
    (begin
      (proc)
      (vamos vamos))))
(define (kosa) (display "cosa"))
```

(Ejemplo **MUY** oscuro).

```
(define call/cc
   call-with-current-continuation)
(define (iter proc)
  (let
    ((vamos (call/cc (lambda(k) k))))
    (begin
      (proc)
      (vamos vamos))))
(define (kosa) (display "cosa"))
(iter kosa)
```

(Ejemplo **MUY** oscuro).

```
(define call/cc
   call-with-current-continuation)
(define (iter proc)
  (let
    ((vamos (call/cc (lambda(k) k))))
    (begin
      (proc)
      (vamos vamos))))
(define (kosa) (display "cosa"))
(iter kosa)
```

Al evaluarse aparece `cosacosacosacosa`$\cdots$

Las continuaciones permiten implementar pseudoparalelismo fácilmente; de hecho, Concurrent ML lo hace así.

Veamos un ejemplo de `yield` de Python, para generar secuencias *lazy*, en C.

Primero, las cabeceras.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <assert.h>
```

Luego algunas definiciones.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <assert.h>
#define TSTACK              4096
#define FIN                  ((int*)1)
typedef void *V, (*PF)(int, int, int);
```

Las continuaciones, con `setjmp` y `longjmp`.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <assert.h>

#define TSTACK            4096
#define FIN               ((int*)1)
typedef void *V, (*PF)(int, int, int);

#define TRANSFER(o, d, r) \
    ({ V _ret_; \
        ((_ret_=(V)setjmp(o))==NULL? \
        ((longjmp(d, (int)r), NULL)): \
        _ret_); })
```

Ahora, el programa. Las variables globales y funciones a usar.

```
jmp_buf t_iter, t_body;
void stack(int iter, jmp_buf buf,
    PF f, int b, int e, int inc)
{
    if(iter==0) {
        alloca(TSTACK);
        stack(!iter, buf, f, b, e, inc);
    }
    else if(setjmp(buf)!=0) {
        f(b, e, inc);
        assert("return en task!");
    }
}
#define ITER(buf, f, b, e, inc) \
    stack(0, buf, f, b, e, inc)
```

Finalmente, cómo se usa esto.

```
void iter(int b, int e, int inc)
{
    static int i;
    for(i=b; i<e; i+=inc)
        TRANSFER(t_iter, t_body, &i);
    TRANSFER(t_iter, t_body, FIN);
}
int main()
{
    ITER(t_iter, iter, 700, 1000, 3);
    for(;;) {
        int *res;
        res=TRANSFER(t_body, t_iter, 1);
        if(res==FIN) break;
        printf("%d\n", *res);
    }
    return 0;
}
```

Otras Protagonistas: las Lógicas Intuicionistas

Surgen como reacción a las paradojas de la Teoría de Conjuntos, principalmente gracias a L. E. J. Brouwer.

Surgen como reacción a las paradojas de la Teoría de Conjuntos, principalmente gracias a L. E. J. Brouwer.

Continuadas por matemáticos y lógicos como Poincaré, Lebesgue, Heyting, Gödel, Gentzen, Kolmogorov, Martin–Löf, etc.

Todas toman como base evitar el uso del Principio del Tercero Excluido, axioma fundamental para las demostraciones por el absurdo. Las lógicas intuicionistas sólo admiten demostraciones constructivas.

Veamos un ejemplo. *Existen irracionales* a *y* b*, tales que* a^b *es racional.*

Veamos un ejemplo. *Existen irracionales a y b, tales que a^b es racional.*

Demostración (por el absurdo): consideremos $\sqrt{2}^{\sqrt{2}}$. Por el principio del tercero excluido, este número será racional o irracional. En el primer caso, basta tomar $a = b = \sqrt{2}$; en el segundo caso, basta tomar $a = \sqrt{2}^{\sqrt{2}}$ y $b = \sqrt{2}$, pues $\sqrt{2}^{\sqrt{2}^{\sqrt{2}}} = 2$.

Veamos un ejemplo. *Existen irracionales a y b, tales que a^b es racional.*

Demostración (por el absurdo): consideremos $\sqrt{2}^{\sqrt{2}}$. Por el principio del tercero excluido, este número será racional o irracional. En el primer caso, basta tomar $a = b = \sqrt{2}$; en el segundo caso, basta tomar $a = \sqrt{2}^{\sqrt{2}}$ y $b = \sqrt{2}$, pues $\sqrt{2}^{\sqrt{2}^{\sqrt{2}}} = 2$.

La demostración es impecable, pero no sabemos si $\sqrt{2}^{\sqrt{2}}$ es racional o no.

Hay otros ejemplos notables, tales como la paradoja de Banach–Tarski, etc.

Un ejemplo de lógica intuicionista es la desarrollada por Gentzen (1.934).

Un ejemplo de lógica intuicionista es la desarrollada por Gentzen (1.934).

Usa como reglas

$$\frac{}{\Gamma \vdash A \Rightarrow A}, ID.$$

$$\frac{\Gamma, B \vdash A}{\Gamma \vdash B \Rightarrow A}, -I.$$

$$\frac{\Gamma \vdash B \Rightarrow A \qquad \Delta \vdash A}{\Gamma, \Delta \vdash A}, -E.$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}, \wedge - I.$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A}, \wedge - E_1.$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash B}, \wedge - E_2.$$

El λ–cálculo

El λ–cálculo es creado por A. Church en la década del 30, con el propósito de servir como fundamento para la lógica y las matemáticas. Es una teoría sumamente básica de las funciones.

Sus componentes son las expresiones λ:

- variables $x, y, \cdots z$,
- abstracciones $\lambda x.E$, y
- aplicaciones $E_1\ E_2$.

Las transformaciones admisibles sobre estos términos son

- α–conversión, $\lambda x.E \equiv \lambda y.E$, si $y \notin FV(E)$;

Las transformaciones admisibles sobre estos términos son

- α–conversión, $\lambda x.E \equiv \lambda y.E$, si $y \notin FV(E)$;
- β–reducción, $(\lambda x.M)\ N \equiv M[N/x]$;

Las transformaciones admisibles sobre estos términos son

- α–conversión, $\lambda x.E \equiv \lambda y.E$, si $y \notin FV(E)$;
- β–reducción, $(\lambda x.M)\ N \equiv M[N/x]$;
- η–abstracción, $M \equiv \lambda x.M$, si $x \notin FV(M)$.

Las transformaciones admisibles sobre estos términos son

- α–conversión, $\lambda x.E \equiv \lambda y.E$, si $y \notin FV(E)$;
- β–reducción, $(\lambda x.M)\ N \equiv M[N/x]$;
- η–abstracción, $M \equiv \lambda x.M$, si $x \notin FV(M)$.

Se dice que una expresión *lambda* está en *forma normal* cuando no se pueden llevar a cabo más β–reducciones.

El intento de basar la lógica en este cálculo falló al ser patente la existencia de expresiones que nunca llegan a una forma normal; ejemplo

$$(\lambda x.xx)(\lambda x.xx).$$

El intento de basar la lógica en este cálculo falló al ser patente la existencia de expresiones que nunca llegan a una forma normal; ejemplo

$$(\lambda x.xx)(\lambda x.xx).$$

Church usó el mismo método que B. Russell empleó para eliminar errores similares en la teoría de Frege: imponer a las expresiones λ un sistema de tipos que impidan construir términos como el anterior.

Las reglas del λ–cálculo simplemente tipado (1.940).

$$\frac{}{x : A \vdash x : A}\, Id$$

$$\frac{\Gamma, x : B \vdash t : A}{\Gamma \vdash \lambda x.t : B \to A}\, {-I}$$

$$\frac{\Gamma \vdash t : B \to A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash t(u) : A}\, {-E}$$

$$\frac{\Gamma \vdash x : A \qquad \Delta \vdash y : B}{\Gamma, \Delta \vdash \langle x, y \rangle : A \wedge B}\, \wedge I$$

$$\frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash fst(t) : A}\, \wedge E_1 \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash snd(t) : B}\, \wedge E_2$$

La β–reducción preserva la derivación de tipos.

Habrán notado cuán parecidas son las reglas de la deducción natural intuicionista con las reglas del λ–cálculo simplemente tipado.

Habrán notado cuán parecidas son las reglas de la deducción natural intuicionista con las reglas del λ–cálculo simplemente tipado. Esto fue notado por H. Curry en 1.956, quien conjeturó una correspondencia entre ambos sistemas, correspondencia finalmente demostrada por Howard en 1.969.

El homomorfismo de Howard–Curry

Proposiciones $\leftrightarrow$ Tipos
Demostraciones $\leftrightarrow$ Programas

El homomorfismo de Howard–Curry

Proposiciones $\leftrightarrow$ Tipos
Demostraciones $\leftrightarrow$ Programas

o, con más precisión,

p es una prueba de la proposición Φ
p es un valor del tipo Φ
p es un programa que cumple la especificación Φ

¡Esto puede ser la respuesta al problema de adecuación de las implementaciones a las especificaciones!

De hecho, varias escuelas han hecho grandes progresos siguiendo esta línea (Alf, Coq, etc.).

¡Esto puede ser la respuesta al problema de adecuación de las implementaciones a las especificaciones!

De hecho, varias escuelas han hecho grandes progresos siguiendo esta línea (Alf, Coq, etc.).

Subsisten serias dificultades, pero es un avance real.

Volvamos a las demostraciones por el absurdo. Es evidente que no pueden servir de base para extraer programas...

Volvamos a las demostraciones por el absurdo. Es evidente que no pueden servir de base para extraer programas...

¿Es realmente evidente?

El fundamento para las demostraciones por el absurdo es el Principio del Tercero Excluido, o lo mismo, la negación de negación:

$$\neg\neg A \equiv A.$$

Informalmente, dos negaciones afirman.

En 1.990, Timothy Griffin estudió una extensión, hecha por M. Felleisen, del λ–cálculo aumentado con dos operaciones, $\mathcal{C}$ (capture) y $\mathcal{A}$ (abort), similares a `callcc` y `throw` de ML.

Al λ–cálculo definido por

$$N ::= x \mid \lambda x.N \mid NN$$

le agrega

$$N ::= \cdots \mid \mathcal{A}(N) \mid \mathcal{C}(N).$$

A fin de darles semántica a $\mathcal{A}$ y $\mathcal{C}$, se explicita el *contexto* en que operan las expresiones λ; si este contexto se denota con $E[X]$, se tiene

$$E[\mathcal{C}(M)] \mapsto_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z]),$$

$$\mathcal{A}(M) \equiv \mathcal{C}(\lambda d.M)$$

con d libre en M. Con esto se tiene que $E[\mathcal{A}(M)] \mapsto_{\mathcal{A}} M$.

Si queremos un `call/cc` à la Scheme podemos tomar

$$(\texttt{call}/\texttt{cc}\ M) = \mathcal{C}(\lambda k.k(Mk)).$$

¿Qué tipos les asociaremos a $\mathcal{C}$ y a $\mathcal{A}$?

¿Qué tipos les asociaremos a $\mathcal{C}$ y a $\mathcal{A}$?
Volvamos a la regla

$$E[\mathcal{C}(M)] \mapsto_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z]).$$

¿Qué tipos les asociaremos a $\mathcal{C}$ y a $\mathcal{A}$?
Volvamos a la regla

$$E[\mathcal{C}(M)] \mapsto_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z]).$$

Imaginemos que E espera un valor de tipo B y $E[-]$ espera otro de tipo A; podemos asignar a $\lambda z.\mathcal{A}(E[z])$ el tipo $A \rightarrow B$

¿Qué tipos les asociaremos a $\mathcal{C}$ y a $\mathcal{A}$?
Volvamos a la regla

$$E[\mathcal{C}(M)] \mapsto_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z]).$$

Imaginemos que E espera un valor de tipo B y $E[-]$ espera otro de tipo A; podemos asignar a $\lambda z.\mathcal{A}(E[z])$ el tipo $A \to B$

Ahora, M debe tener como tipo $(A \to B) \to B$; concluímos que $\mathcal{C}$ debe tener tipo A.

Si N es in término de tipo B, a $\mathcal{A}(N) = \mathcal{C}(\lambda z.N)$ le podemos dar *cualquier tipo* A.

Si N es in término de tipo B, a $\mathcal{A}(N) = \mathcal{C}(\lambda z.N)$ le podemos dar *cualquier tipo* A.

Ahora, si queremos interpretar los tipos como proposiciones, B debe corresponder a una proposición *sin demostración*. Podemos asignarle $\perp$.

Si N es in término de tipo B, a $\mathcal{A}(N) = \mathcal{C}(\lambda z.N)$ le podemos dar *cualquier tipo* A.

Ahora, si queremos interpretar los tipos como proposiciones, B debe corresponder a una proposición *sin demostración*. Podemos asignarle $\bot$.

Definimos

$$\neg A \equiv A \to \bot.$$

De aquí, tenemos que, si M tiene tipo $\neg\neg A$, $\mathcal{C}(M)$ tiene tipo A. Lógicamente, corresponde a la regla de eliminación de doble negación

$$\frac{\neg\neg A}{A}\,.$$

Extensiones y profundizaciones fueron hechas por Ch. Murthy, Parigot, J. Krivine, A. Filinski, H. Thielecke, etc.

No todas son rosas: hay demostraciones que no terminan; esto es, no generan programas.

Los programas clásicos no son observacionalmente congruentes con su traducción CPS.

Un ejemplo: la sucesión de Fibonacci, H. Schwichtenberg, usando MINLOG.

Un ejemplo: la sucesión de Fibonacci, H. Schwichtenberg, usando MINLOG.

Usaremos el cuantificador existencial clásico:

$$\exists x.p(x) \equiv_{def} \neg(\forall x.\neg p(x)).$$

De la prueba de

$$\forall n \exists^{cl} k G(n, k),$$

De la prueba de

$$\forall n \exists^{cl} k G(n, k),$$

se extrae el siguiente algoritmo (mostrado en Scheme):

```
(define (fibo n)
 (fibo1 n (lambda (k 1) k)))
(define (fibo1 n1 f)
 (if (= n1 0)
   (f 0 1)
   (fibo1 (- n 1)
     (lambda (k 1) (f 1 (+ k 1))))))
```

Ahora, con sinceridad...

Ahora, con sinceridad...
Lo anterior, qué puede ser sino

Ahora, con sinceridad...
Lo anterior, qué puede ser sino

LA VENGANZA DEL GOTO

Bibliografía y URLs de interés

John Reynolds (1.997), *Theories of Programming Languages* Cambridge University Press.

Andrew Appel (1.991), *Compiling with Continuations* Cambridge University Press.

Timothy G. Griffin, *A formmulæ–as–types notion of control* Conference Record of the Seventeenth Anual ACM Symposium on Principles of Programming Languages (1.990).

Chetan Murthy, *Extracting constructive content from classical proofs* Technical Report 90–1151, Dep. of Comp. Science, Cornell Univ., Ithaca, NY.

Helmut Schwichtenberg, *Program Extraction from Proofs* Mathematisches Institut der Universität München, Sommersemester,2002.

Phil Wadler Home Page.

Andrzej Filinski Home Page.

Hayo Thielecke Home Page.

¿Preguntas?

¡Muchas gracias!