# La Evaluación Parcial: qué es y para qué sirve.

Pascual Julián Iranzo*
Departamento de Informática, U. de Castilla-La Mancha,
Ronda de Calatrava s/n, 13071 Ciudad Real, España.
pjulian@inf-cr.uclm.es

**Resumen**

La *evaluación parcial* 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*. Este artículo trata de presentar la evaluación parcial a un público no experto; con este fin se traza una panorámica general del área y de sus técnicas más importantes. Si bien se define con precisión qué se entiende por evaluación parcial y se ilustran sus ventajas mediante diversos ejemplos, este trabajo se ocupa, principalmente, de los objetivos de la evaluación parcial y de sus aplicaciones, particularmente en el campo de la inteligencia artificial.

**Palabras clave:** transformación de programas; evaluación parcial; especialización de programas; optimización de programas; generación automática de programas; aplicaciones de la evaluación parcial; supercompilación. **Palabras clave adicionales:** inteligencia artificial; metacomputación; compiladores e intérpretes; mantenimiento del software; verificación de programas.

## 1. Introducción.

La transformación de programas es un método para derivar programas correctos y eficientes partiendo de una especificación ingenua y más ineficiente del problema. Esto es, dado un programa $P$, se trata de generar un programa $P'$ que resuelve el mismo problema y equivale semánticamente a $P$, pero que goza de mejor comportamiento respecto a cierto criterio de evaluación. En la literatura se ha propuesto una gran variedad de transformaciones para mejorar el código. Una de las mejor estudiadas es la *Evaluación Parcial* (EP) de programas (también conocida como *especialización*), que ofrece un marco unificado para la investigación acerca de los procesadores de lenguajes, en particular, compiladores e intérpretes [36, 38]. La EP es una técnica de transformación de programas que consiste en la especialización de programas respecto a ciertos datos de entrada, conocidos en tiempo de compilación [25]. En general, las técnicas de EP incluyen algún criterio de parada para garantizar la terminación del proceso de la transformación [43, 48]. La EP es, por tanto, una técnica de transformación automática, lo cual la distingue de otras técnicas de transformación de programas tradicionales [16, 51]. La EP ha sido aplicada a los lenguajes imperativos tanto como a los declarativos y a una gran variedad de problemas concretos. Una panorámica general sobre este campo y su área de aplicación se presenta en [38] y en [39]. Un breve pero excelente tutorial sobre EP (si bien centrado en la especialización de programas imperativos y funcionales) es [21]. Otras obras que examinan aspectos concretos del área son [7, 26].

*Miembro del grupo ELP (Extensiones de la Programación Lógica) del DSIC. Universidad Politécnica de Valencia. Camino de Vera s/n VALENCIA.

Este artículo se estructura como sigue. En el Apartado 2, se describe con precisión el concepto de EP, se introducen algunas de sus principales técnicas y se plantea el problema de la corrección del método, un punto de gran interés en su fundamentación teórica; también se estudia la relación existente entre la EP y la generación automática de programas. En el Apartado 3 se presentan algunos de los objetivos más importantes perseguidos por la EP, como son: el aumento de la eficiencia de los programas; el aumento de productividad en el desarrollo de las aplicaciones; mejorar la reusabilidad y modularidad del software y conseguir evaluadores parciales autoaplicables (i.e, evaluadores parciales capaces de especializarse a ellos mismos). En el Apartado 4 se pone especial cuidado en exponer varias de las aplicaciones de la EP en áreas industriales y en otros campos de investigación más básica. En el Apartado 5 se facilitan una serie de direcciones URL donde encontrar información general sobre el campo de la EP y sobre grupos de investigación que desarrollan una actividad intensa en este área. Este apartado da pie a comentar algunos de los evaluadores parciales desarrollados para diferentes lenguajes; tanto declarativos como imperativos. El artículo termina con una discusión sobre el papel de la EP en la metaprogramación y finalmente se presentan las conclusiones.

## 2. La Evaluación Parcial.

La idea de especializar funciones con respecto a uno o varios de sus argumentos es vieja en el campo de la teoría de funciones recursivas, donde recibe el nombre de *proyección*. Esta posibilidad está implícita en el teorema s-m-n de Kleene [41] que establece que, dada una función computable $f \in \mathcal{F}^{n+m}$ tal que $f : S^{n+m} \to S$, se cumple que:

1. la función $f_{e_1,\ldots,e_n} : S^m \to S$, tal que

$$f_{e_1,\ldots,e_n}(d_1,\ldots,d_m) = f(e_1,\ldots,e_n,d_1,\ldots,d_m),$$

es computable;

2. la función de orden superior $f^* : \mathcal{F}^{n+m} \times S^n \to \mathcal{F}^m$, tal que

$$f^*(f,e_1,\ldots,e_n) = f_{e_1,\ldots,e_n},$$

es computable;

**Ejemplo 1** *Dada la función suma* $: I\!\!N^2 \to I\!\!N$, *definida como* $suma(x,y) = x + y$, *puede especializarse para un valor conocido de* $x$, *por ejemplo 2, transformándose en una función* $suma_2 : I\!\!N \to I\!\!N$, *tal que* $suma_2(y) = 2 + y$.

Se debe notar que los valores conocidos simplemente se substituyen por los correspondientes parámetros formales de la función; no se realiza ningún cómputo. Además, en este proceso, que recuerda a la *currificación* de funciones en los lenguajes funcionales, el objetivo de lograr una mayor eficiencia está ausente. Los problemas de eficiencia eran irrelevantes para las investigaciones efectuadas por Kleene, centradas en establecer los límites entre lo que es o no es computable. El hecho es que la técnica propuesta por Klenee produce funciones especializadas cuyo comportamiento operacional, en algunos casos, es "peor" que el de las originales. Por otro lado, la evaluación parcial trabaja con programas (textos) más bien que con funciones matemáticas. Por consiguiente, la EP es una técnica que va más allá de la simple proyección de funciones matemáticas.

La EP establece como ejecutar un programa cuando sólo conocemos parte de sus datos de entrada. De forma más precisa, dado un programa $P$ y parte de sus datos de entrada $in1$, el objetivo de la EP es construir un nuevo programa $P_{in1}$ que cuando recibe el resto de los datos de entrada $in2$, computa el mismo resultado que produce $P$ al procesar toda su entrada ($in1$+$in2$). Esto último asegura la corrección de la transformación efectuada. El programa que realiza el proceso de EP recibe el nombre de *evaluador parcial* y el resultado de la EP, el programa $P_{in1}$, se denomina *programa especializado*, *programa evaluado parcialmente* o también *programa residual*. La idea que se esconde detrás del proceso de EP consiste en: i) realizar tantos cómputos como sea posible en tiempo de EP, haciendo uso de los datos de entrada conocidos $in1$, también denominados *datos estáticos* (por contraposición con los datos de entrada desconocidos $in2$, que son denominados *datos dinámicos*, y que sólo se conocen en tiempo de ejecución del programa residual); ii) generar código relacionado con aquellos cálculos que no puedan realizarse por depender de los datos de entrada desconocidos. Así pues, un evaluador parcial realiza una mezcla de acciones de cómputo y de generación de código; esta es la razón por la que Ershov denominó a la EP *computación mixta* (*mixed computation*). Cuando se realizan cómputos en tiempo de EP, haciendo uso de los datos de entrada conocidos, decimos que hay *propagación de la información*. El objetivo de la EP es obtener la mejor de las especializaciones posibles maximizando la propagación de la información. Se espera que el programa resultante pueda ejecutarse de forma más eficiente ya que, usando el conjunto de datos (parcialmente) conocidos, es posible evitar algunas computaciones (en tiempo de ejecución) que se realizarán (una única vez) durante el proceso de EP. Para cumplir estos fines, la EP utiliza, además de la computación simbólica, algunas técnicas bien conocidas provenientes de la transformación de programas (funcionales) [16], procurando su automatización:

1. Definición: permite la introducción de funciones nuevas o la extensión de las existentes.

2. Instanciación: permite especializar una función asignando datos de entrada conocidos a sus argumentos.

3. Desplegado (*unfolding*): permite el reemplazamiento de una llamada a función por su respectiva definición.

4. Plegado (*folding*): como su nombre indica, es la transformación inversa del desplegado, es decir, el reemplazamiento de cierto fragmento del código por la correspondiente llamada a función

La Figura 1 concreta estas técnicas de transformación para el caso en el que las funciones están definidas mediante reglas de reescritura[1]. Las reglas de inferencia de la Figura 1 deben utilizarse de modo semejante al empleado en la lógica. Así pues, al aplicar la regla de `definición` tendremos la posibilidad de introducir una regla nueva $l_2 \to r_2$ en un programa $\mathcal{R}$, si no existe ninguna regla $l_1 \to r_1 \in \mathcal{R}$ tal que $l_1$ solape con $l_2$. La aplicación de cada una de estas reglas da lugar a una secuencia de transformación de la que, finalmente, se extraerá el programa transformado. Una estrategia, que reduce el alto grado de indeterminismo existente en la aplicación de las reglas de la Figura 1, es la presentada en el siguiente esquema básico de transformación:

1. Repetir hasta que convenga

- buscar una `definición`, e
- `instanciar` la definición para permitir,
- pasos de `desplegado` en diversos puntos,

[1]En la Figura 1 se muestran solamente aquellas reglas que presentan un interés inmediato para las técnicas de EP.

1. `Definición`:

$$\frac{(\forall R_i)(R_i \equiv (l_1 \to r_1) \in \mathcal{R} \wedge \neg unifica(l_1, l))}{(l \to r) \in \mathcal{R}}$$

2. `Instanciación`:

$$\frac{(l \to r) \in \mathcal{R}}{(\theta(l) \to \theta(r)) \in \mathcal{R}}$$

3. `Desplegado` (*unfolding*):

$$\frac{(l_1 \to r_1) \in \mathcal{R} \wedge (l_2 \to C[\theta(l_1)]) \in \mathcal{R}}{(l_2 \to C[\theta(r_1)]) \in \mathcal{R}}$$

4. `Plegado` (*folding*):

$$\frac{(l_1 \to r_1) \in \mathcal{R} \wedge (l_2 \to C[\theta(r_1)]) \in \mathcal{R}}{(l_2 \to C[\theta(l_1)]) \in \mathcal{R}}$$

donde: $\theta$ es una substitución (i.e., una función que asigna a cada variable un término); $C[\Box]$ es un contexto y $C[t]$ es el resultado de substituir en el contexto $C[\Box]$ cada una de las apariciones del símbolo "$\Box$" por el término $t$.

Figura 1: Reglas de transformación de Burstall y Darlington.

seguidos por,

- un `plegado` de las reglas resultantes haciendo uso de alguna de las reglas obtenidas.

2. `Extraer` el programa transformado.

De entre todas estas técnicas, el desplegado es la herramienta de transformación fundamental de la EP. Para programas funcionales, los pasos de plegado y desplegado sólo involucran ajuste de patrones (*pattern matching*). En el caso de los programas lógicos, el ajuste de patrones se substituye por la unificación, obteniéndose así una mayor potencia de propagación de la información. Una técnica específica empleada en la EP es la denominada *especialización de puntos de control* del programa, que combina las reglas de definición, desplegado y plegado. Esta técnica puede entenderse como un proceso consistente en una definición al que le sigue una sucesión de pasos de desplegado (tantos como sea posible) que se detienen en cuanto se reconoce una configuración "ya vista" anteriormente, momento en el que se genera código para llamar a esa configuración "ya vista" (i.e., se realiza un paso de plegado). En un lenguaje imperativo, un *punto de control* es una etiqueta del programa; en un lenguaje declarativo puede considerarse que es la definición de una función o predicado. La idea es que una etiqueta o una función del programa $P$ pueda aparecer en el programa especializado $P_{in_1}$ en varias versiones especializadas, cada una correspondiente a datos determinados conocidos en tiempo de EP. Es conveniente notar que esta técnica es un reflejo del esquema básico de transformación anteriormente esbozado. Otra de las técnicas empleadas por la EP es la *abstracción*[2], consistente en generalizar una expresión cuando no es posible su especialización; en cierto sentido, puede verse como la transformación inversa de la instanciación y puede caracterizarse en términos de un proceso de definición al que le sigue uno de plegado [4].

A continuación ilustramos el proceso de EP y aclaramos algunos de los conceptos y técnicas introducidos, mediante una serie de ejemplos.

**Ejemplo 2** *Sea el fragmento de un programa $P$ que computa la función $x^n$:*

$pow(0, X) \to 1$
$pow(N, X) \to X * pow(N-1, X)$

*Si queremos especializar el programa para el dato de entrada conocido $N = 3$, los pasos que podría realizar un hipotético evaluador parcial serían:*

```
% Definición
```
$pow3(X) \to pow(3, X)$
```
% Instanciación, desplegado y
  computación simbólica
```
$pow(3, X) \to X * pow(3-1, X)$
$pow(3, X) \to X * pow(2, X)$
$pow(3, X) \to X * (X * pow(2-1, X))$
$pow(3, X) \to X * (X * pow(1, X))$
$pow(3, X) \to X * (X * (X * pow(1-1, X)))$
$pow(3, X) \to X * (X * (X * pow(0, X)))$
$pow(3, X) \to X * (X * (X * 1))$
$pow(3, X) \to X * (X * X)$
```
% Desplegado final
```
$pow3(X) \to X * (X * X)$

*conduciendo al programa especializado $P_3$:*

$pow3(X) \to X * (X * X)$

*que computa la función $x^3$.*

De manera simple, podemos entender que $pow3(X)$ constituye una especialización del punto de control $pow(N, X)$ en la que no ha sido necesario realizar pasos de plegado, debido a que el programa especializado no contiene llamadas a función. Tampoco se han requerido pasos de abstracción para lograr la especialización.

**Ejemplo 3** *Consideremos de nuevo el programa del Ejemplo 2. Si queremos especializar el programa con respecto a la expresión $(pow(M, B) * pow(N, B))$, los pasos que podría realizar un hipotético evaluador parcial serían:*

```
% Definición
```
$ppow(M, N, B) \to pow(M, B) * pow(N, B)$
```
% Instanciación, desplegado y
  computación simbólica
```
$ppow(0, N, B) \to pow(0, B) * pow(N, B)$
$ppow(0, N, B) \to 1 * pow(N, B)$
$ppow(0, N, B) \to pow(N, B)$
```
% Instanciación, desplegado y
  computación simbólica
```
$ppow(M, 0, B) \to pow(M, B) * pow(0, B)$
$ppow(M, 0, B) \to pow(M, B) * 1$
$ppow(M, 0, B) \to pow(M, B)$
```
% Desplegado y computación simbólica
```
$ppow(M, N, B) \to B * pow(M-1, B) * pow(N, B)$
$ppow(M, N, B) \to$
$\quad B * pow(M-1, B) * B * pow(N-1, B)$
$ppow(M, N, B) \to$
$\quad B * B * pow(M-1, B) * pow(N-1, B)$
```
% Plegado
```
$ppow(M, N, B) \to B * B * ppow(M-1, N-1, B)$

*Pudiendo extraerse el programa especializado:*

$ppow(0, N, B) \to pow(N, B)$
$ppow(M, 0, B) \to pow(M, B)$
$ppow(M, N, B) \to B * B * ppow(M-1, N-1, B)$
$pow(0, X) \to 1$
$pow(N, X) \to X * pow(N-1, X)$

*que permite el cómputo de la expresión $(pow(M, B) * pow(N, B))$ de manera más eficiente que el programa original.*

La definición de la función $ppow(M, N, B)$ constituye una especialización del punto de control $pow(N, X)$ que facilia el cómputo de la expresión $(pow(M, B) * pow(N, B))$. En este ejemplo se ha podido realizar un paso de plegado, debido a que en el proceso de especialización hemos reconocido la aparición de una regularidad: la expresión "ya vista" $pow(M-1, B) * pow(N-1, B)$, que puede substituirse por su definición $ppow(M-1, N-1, B)$, dando lugar a un paso de desplegado. El programa especializado, si bien no obtiene una mejora en cuanto al número de multiplicaciones a realizar, consigue fundir las secuencias de llamadas recursivas iniciadas por $(pow(M, B)$ y $pow(N, B))$, que deberían realizarse por separado, en una única secuencia en la que se simplifica el control. Si se hubiese partido de un programa iterativo, el efecto hubiese

[2] La abstracción es una técnica compleja cuya caracterización precisa está fuera del alcance pretendido en esta introducción

sido la combinación de dos bucles en uno solo.

Aunque al comentar los ejemplos 2 y 3 no se ha hecho énfasis en el control del proceso de EP, estos ejemplos ilustran una forma de EP, denominada *online*, en la que todas las decisiones de control (e.g. ¿qué evaluar?, ¿cuándo parar?) se toman en tiempo de EP. El próximo ejemplo sirve para ilustrar una aproximación diferente a la EP *online*.

**Ejemplo 4** *Consideremos de nuevo la función del Ejemplo 2 pero expresada en un lenguaje imperativo como el lenguaje C.*

```
int pow(int n, int x)
{
 int power = 1;
 while (n > 0) {
 power = power * x;
 n = n - 1;
 };
 return power;
}
```

*Un evaluador parcial típico de un lenguaje imperativo necesita que se le proporcione como entrada el texto del programa así como ciertas especificaciones que le permitan distinguir entre los datos estáticos y los dinámicos. El usuario podría indicar que el primer argumento es estático y el segundo dinámico. Con estas indicaciones el evaluador parcial realiza un análisis del programa en el que introduce una serie de anotaciones, indicando qué partes del código del programa original serán evaluadas en tiempo de EP. En el código que se muestra a continuación estas anotaciones aparecen en negrita.*

```
int pow(int n, int x)
{
 int power = 1;
 while (n > 0) {
 power = power * x;
 n = n - 1;
 };
 return power;
}
```

*Una vez realizado este análisis, el usuario debe suministrar valores a los argumentos estáticos, e.g. asignar el valor* 3 *al argumento* n *de la función. Entonces, el evaluador parcial obtiene el siguiente programa residual:*

```
int pow3(int x)
{
 int power = 1;
 power = power * x;
 power = power * x;
 power = power * x;
 return power;
}
```

*Los evaluadores parciales también permiten un postproceso de* compresión de código *que conduciría al programa transformado óptimo:*

```
int pow3(int x)
{
 int power;
 power = 1 * x * x * x;
 return power;
}
```

La clasificación de las variables del programa en estáticas o dinámicas recibe el nombre de *división*. Esta tarea es más compleja de lo que podría suponerse a simple vista, siendo difícil de implementar de forma automática, por lo que suele requerir la intervención humana. El proceso de computar una división adecuada partiendo de una división inicial provisional de las variables que se consideran la entrada del programa, recibe el nombre de *binding-time analysis* pues en él se determina el momento en el cual puede computarse el valor de una variable, i.e., cuándo un valor se enlaza a la variable. En el Ejemplo 4 se ha descrito la EP como un proceso consistente en varias fases: *binding-time analysis*, generación de anotaciones y EP propiamente dicha. Este tipo de aproximación se denomina EP *offline*, por contraposición con la EP *online*. Debido a que el proceso de *binding-time analysis* es siempre aproximado y porque en un evaluador parcial *online* la decisión de qué expresión debe evaluarse se toma en tiempo de EP, lo que supone una ventaja, los evaluadores parciales *online* permiten alcanzar una mayor precisión en la especialización de los programas que los evaluadores parciales *offline*. En los evaluadores parciales *offline* es crítico encontrar la división adecuada ya que ésta determina el grado de especialización que se alcanzará [21].

Un tema importante relativo a la EP es su capacidad para reestructurar los puntos de control. La reestructuración de puntos de control tiene que ver con las relaciones que se establecen entre los puntos de control del programa original y los del programa residual. En la nomenclatura de [30] podemos distinguir las siguientes capacidades de reestructuración:

- *Monovariante*: cualquier punto de control del programa original da lugar, como mucho, a un punto de control en el programa residual; "como mucho" quiere decir que un punto de control del programa original puede desaparecer (i.e., puede dar lugar a cero puntos de control) en el programa residual. El Ejemplo 2 muestra una reestructuración monovariante, en la que el punto de control $pow(N, X)$ del programa original da lugar a un único punto de control especializado $pow3(X)$ en el programa residual.

• *Polivariante*: cualquier punto de control del programa original puede dar lugar a uno o más puntos de control en el programa residual. El Ejemplo 3 muestra una reestructuración polivariante, en la que el punto de control $pow(N, X)$ del programa original da lugar a dos puntos de control, $pow(N, X)$ y $ppow(M, N, X)$, en el programa residual.

• *Monogenético*: cualquier punto de control del programa residual se produce a partir de un único punto de control del programa original. El Ejemplo 2 muestra una reestructuración monogenética, en la que punto de control especializado $pow3(X)$, en el programa residual, se produce a partir del punto de control $pow(N, X)$ del programa original.

• *Poligenético*: cualquier punto de control del programa residual se produce a partir de uno o más puntos de control del programa original. Ejemplo 3 muestra una reestructuración poligenética, en la que el punto de control especializado $ppow(M, N, X)$, en el programa residual, combina varias definiciónes de función del programa original: la función $pow(N, X)$ y el operador "$*$".

Una buena técnica de EP debe ofrecer una capacidad de reestructuración tanto polivariante como poligenética. En [1, 40] se detalla un algoritmo de EP *online*, capaz de producir especialización polivariante y poligenética, que engloba la EP de programas lógicos y la EP de programas funcionales.

## 2.1. Corrección de la Evaluación Parcial.

En cuanto a la fundamentación teórica de la EP, el tema más relevante es el de la *corrección* de la misma. A este respecto hay dos puntos a tratar: la corrección semántica (i.e., la preservación del comportamiento observable) y el control de la terminación.

Comenzaremos discutiendo la corrección semántica de la evaluación parcial. Con el fin de formalizar este concepto en un marco general, se introducen las siguientes convenciones y notaciones estándares [38], que serán útiles en este subapartado y en el próximo. Vamos a tratar con diversos lenguajes; emplearemos las letras $L, S$ y $T$ para referirnos, respectivamente, a un lenguaje de implementación, a un lenguaje fuente y a un lenguaje objeto. Denotaremos por $D$ el conjunto de los datos que pueden pasarse como valores a un programa, incluyendo también los textos que forman los programas. Supondremos que suministramos listas de datos como entrada para los programas; denotaremos por $D^*$ el conjunto de todas las listas que pueden formarse a partir de los elementos de $D$. Suponemos que la semántica de los lenguajes que empleamos está definida en un estilo operacional no especificado. Si el lenguaje es imperativo, suponemos que los cómputos se realizan mediante una secuencia de instrucciones que producen cambios de estado. Más genéricamente, podemos asumir que los cómputos se realizan mediante deducción aplicando ciertas reglas de inferencia. Si $P$ es un programa en un lenguaje $L$, entonces $[\![P]\!]_L$ denota el significado del programa $P$ escrito en el lenguaje $L$. El significado del programa se establece como una función $[\![P]\!]_L : D^* \rightarrow D \cup \{\bot\}$ tal que si $in \in D^*$ entonces

$$out = [\![P]\!]_L(in)$$

siendo $out \in D$ el resultado de ejecutar $P$ para la entrada $in$; cuando la ejecución del programa $P$ no termina el valor de $out$ es $\bot$ (indefinido).

Supongamos un programa fuente $P$ al que se suministra la entrada de datos estática $e$, conocida previamente, y la entrada de datos dinámica $d$, conocida con posterioridad. Entonces definimos la EP de $P$, utilizando un evaluador parcial $Spec$ mediante la ecuación:

$$P_e = [\![Spec]\!]_L([P, e]).$$

**Definición 2.1**
*Supuesto que el proceso de EP termina, decimos que un evaluador parcial es:*

1. **Correcto** *si y sólo si para toda entrada dinámica $d \in D$, $[\![P_e]\!]_T(d) = out$ implica que $[\![P]\!]_S([e, d]) = out$ (i.e., si $out$ es un valor computado por $P_e$ entonces $P$ lo computa también.).*

2. **Completo** *si y sólo si para toda entrada dinámica* $d \in D$*,* $[\![P]\!]_S([e,d]) = out$ *implica que* $[\![P_e]\!]_T(d) = out$ *(i.e., si out es un valor computado por* $P$ *entonces* $P_e$ *lo computa también.).*

Los conceptos de corrección y completitud se han definido en un sentido *fuerte* de forma que, si un evaluador parcial es correcto y completo, podemos establecer la identidad semántica entre el programa especializado y el original. Esto es, se cumple que

$$[\![P_e]\!]_T(d) = [\![P]\!]_S([e,d]),$$

para cada entrada dinámica $d \in D$. En palabras, $P$ y $P_e$ computan los mismos valores (salidas).

El control de la terminación es un problema crucial en el ámbito de la EP. La no terminación es un comportamiento indeseable para una herramienta que pretende optimizar programas automáticamente. La no terminación del proceso de EP puede producirse por una de las siguientes razones:

1. Un intento de construir una instrucción, una expresión, o en general una estructura infinita.

2. Un intento de construir un programa residual que contenga infinitos puntos de control (etiquetas, procedimientos, funciones definidas o predicados).

La causa última de este comportamiento es la misma: un fallo en la estrategia de desplegado que emplea el evaluador parcial [37]. En cualquier caso, la no terminación hace que un evaluador parcial sea poco aceptable para su uso por parte de un usuario inexperto, y completamente inaceptable para su uso como herramienta de generación automática de software. Para asegurar la terminación de la evaluación parcial, durante el proceso de especialización la mayoría de los evaluadores parciales mantienen una historia de la computación, que permite su consulta a la hora de tomar decisiones sobre si desplegar una expresión o no; y, en caso de que la decisión sea "no", determinar cómo realizar el plegado para especializar la expresión o si hay que abstraer, para convertirla en otra más general. Existen varios compromisos que deben tenerse en consideración cuando realizamos la EP: desplegar con suma liberalidad puede llevar al problema (1), dando lugar a un tiempo de especialización infinito y a la no generación del programa residual; generar expresiones residuales demasiado especializadas (i.e., conteniendo demasiados datos estáticos) puede conducir a un programa residual infinito y, por lo tanto, a la aparición del problema (2); en el otro extremo, generar expresiones residuales demasiado generales puede hacer perder toda la especialización, obteniéndose programas residuales que son (esencialmente) el programa original sin especializar. Los aspectos que acabamos de comentar han permitido distinguir dos niveles de control en el problema de la terminación de la EP [49] que, en buena medida, pueden ser abordados de forma independiente: el llamado *control local*, asociado con el punto (1); y el llamado *control global*, asociado con el punto (2).

Un aspecto relacionado con el problema de la terminación es el de la *precisión*, entendiendo por tal la obtención del máximo potencial especializador. Aquí, también podemos distinguir dos niveles: i) el nivel de *precisión local*, asociado con el desplegado de una expresión y el hecho de que puede perderse potencial para la especialización si se detiene el desplegado de la expresión demasiado pronto (o demasiado tarde); y ii) un nivel de *precisión global* que está asociado al número de puntos de control especializados; en general disponer de un programa residual con el mayor número de puntos de control especializados con respecto a una variedad de datos estáticos conducirá a una mejor especialización.

Como puede apreciarse, el control de la EP ofrece aspectos que pueden entrar en conflicto, como son el de la terminación y el de la precisión. Un buen algoritmo de EP debe asegurar la corrección y la terminación mientras minimiza las pérdidas de precisión.

Por último, se dice que un especializador es *parcialmente* correcto si, en el supuesto de que termina, genera un programa residual semánticamente equivalente al original. Si además de generar un programa residual semánticamente equivalente al original, termina cualquiera que sea la circunstancia, se habla de *corrección total*.

## 2.2. Evaluación Parcial y Generación Automática de Programas.

En este apartado formalizamos el concepto de evaluador parcial, lo que nos permite establecer relaciones interesantes entre la evaluación parcial y la generación automática de programas.

Para establecer la propiedad esencial que caracteriza a un evaluador parcial de una manera formal, supongamos un programa fuente $P$ al que se suministra la entrada de datos estática $in_1$ y la entrada de datos dinámica $in_2$. Entonces podemos describir el cómputo (del resultado) en un paso mediante la ecuación,

$$out = [\![P]\!]_S([in_1, in_2])$$

y el cómputo (del resultado) en dos pasos, utilizando un evaluador parcial *Spec* mediante las ecuaciones

$$\begin{array}{rcl} P_{in_1} & = & [\![Spec]\!]_L([P, in_1]) \\ out & = & [\![P_{in_1}]\!]_T(in_2). \end{array}$$

Combinando estas ecuaciones se obtiene una *definición ecuacional* del evaluador parcial *Spec*:

$$[\![P]\!]_S([in_1, in_2]) = [\![[\![Spec]\!]_L([P, in_1])]\!]_T(in_2)$$

donde, si una parte de la ecuación está definida, también lo estará la otra y con el mismo valor. De estas ecuaciones también se desprende que el evaluador parcial puede estar escrito en un lenguaje de implementación $L$, tener como entrada un programa $P$ escrito en un lenguaje fuente $S$, y producir como salida un programa especializado escrito en un lenguaje objeto[3] $T$. Cuando solamente se emplea un lenguaje en la discusión, los subíndices pueden eliminarse obteniéndose la siguiente ecuación simplificada:

$$[\![P]\!]([in_1, in_2]) = [\![[\![Spec]\!]([P, in_1])]\!](in_2)$$

Si el evaluador parcial, escrito en un lenguaje $L$, admite como entrada programas también escritos en $L$, se dice que es *autoaplicable*. La capacidad de autoaplicación posibilita la escritura de evaluadores parciales que puedan especializarse a sí mismos; éste es un tema de gran interés dentro del campo de la EP.

[3]Posiblemente un lenguaje máquina, si el interés primordial fuese el incremento de la eficiencia del programa especializado con respecto al original.

La definición ecuacional del evaluador parcial nos permite poner en relación los conceptos de interpretación, compilación y EP. Primeramente, nótese que un intérprete es un programa $Int$, escrito en un lenguaje $L$, que puede ejecutar un programa $Fuente$ en un lenguaje $S$ junto con sus datos de entrada $in$. En símbolos,

$$[\![Fuente]\!]_S(in) = [\![Int]\!]_L([Fuente, in]),$$

que es la ecuación de definición de un *intérprete* $Int$ para $S$ escrito en $L$. Un compilador es un programa $Comp$, escrito en un lenguaje $L$, que genera un programa $Objeto$ en un lenguaje $T$. En símbolos,

$$Objeto = [\![Comp]\!]_L(Fuente),$$

que es la ecuación que define un programa $Objeto$. El efecto de ejecutar el programa $Fuente$ sobre los datos de entrada $in$ se consigue, una vez realizada la compilación, ejecutando el programa $Objeto$ con los datos de entrada $in$,

$$[\![Fuente]\!]_S(in) = [\![Objeto]\!]_T(in).$$

Combinando estas ecuaciones obtenemos la definición de un *compilador* $Comp$ de $S$ a $T$ escrito en $L$,

$$[\![Fuente]\!]_S(in) = [\![[\![Comp]\!]_L(Fuente)]\!]_T(in)$$

Ahora pueden resumirse las capacidades de la EP para la *generación automática* de programas mediante la siguiente proposición.

**Proposición 2.2** (**Proyecciones de Futamura**) *Sean Spec un especializador de $L$ a $T$ escrito en $L$, e Int un intérprete para $S$ escrito en $L$.*

- $1^a$ *proy.* : $Objeto = [\![Spec]\!]_L([Int, Fuente])$
- $2^a$ *proy.* : $Comp = [\![Spec]\!]_L([Spec, Int])$
- $3^a$ *proy.* : $Cogen = [\![Spec]\!]_L([Spec, Spec])$

Estas ecuaciones, aunque sencillas de probar (ver [38]), no son fáciles de entender intuitivamente. La primera de ellas indica que se puede compilar un programa $Fuente$ especializando su intérprete $Int$ con respecto a dicho programa $Fuente$. La segunda dice que se puede obtener un compilador mediante autoaplicación del evaluador parcial, i.e., especializando el propio $Spec$ con respecto a un intérprete $Int$. La tercera establece que $Cogen$ es un generador de compiladores, que transforma un intérprete en un compilador, i.e., $Comp = [\![Cogen]\!]_T(Int)$. Así pues, la EP permite la compilación (primera proyección), la generación de compiladores (segunda proyección) y la generación de generadores de compiladores (tercera proyección).

# 3. Objetivos de la Evaluación Parcial.

En este apartado se discuten varios de los objetivos más importantes de la EP.

## 3.1. Aumento en la Eficiencia de los Programas.

Una de las principales motivaciones de la EP es el aumento en la *eficiencia* (*speedup*) de los programas. Debido a que parte de los cómputos se han realizado previamente, en tiempo de EP, esperamos que el programa especializado sea más rápido que el programa original. Es común realizar una medida del aumento de la eficiencia, obteniendo la razón entre el tiempo de ejecución del programa original y del especializado [38, 37]. A la hora de medir la eficiencia de la EP, debe considerarse también el coste del tiempo de especialización del programa original. La EP es claramente ventajosa cuando un (procedimiento de un) programa debe ejecutarse reiteradamente para una porción de su entrada, ya que entonces el coste que pueda suponer la especialización del programa será ampliamente amortizado por las sucesivas ejecuciones del programa especializado, que será más "rápido" que el original. Neil D. Jones argumenta en [38] que la EP puede ser ventajosa incluso para una única ejecución, ya que muchas veces sucede que el tiempo invertido en la especialización del programa original sumado al tiempo de ejecución del programa especializado es inferior al tiempo que resultaría si se ejecutase el programa original con toda su entrada.

Como ejemplo, puede afirmarse que el evaluador parcial de programas declarativos multiparadigma Indy (ver Apart. 5) consigue mejoras en la eficiencia de los programas especializados que en muchos casos superan el 100 %.

## 3.2. Productividad, Reusabilidad y Modularidad.

Otro objetivo de la EP es propiciar la *productividad* en el desarrollo de programas mediante el aumento de la *reusabilidad* y la *modularidad* del código. De todos es sabido que es más fácil establecer el significado declarativo (correcto) para una especificación sencilla de un problema; por contra, la ejecución de esta especificación como programa puede resultar ineficiente. Un evaluador parcial puede facilitar y hacer más ágil el desarrollo de los programas al permitir explotar una biblioteca de plantillas genéricas y simples (cuyo significado declarativo fuese, sin duda, el esperado) que posteriormente se especializan de forma automática para producir un código más eficiente. La corrección del proceso de EP asegura la corrección semántica del programa especializado. Disponer de una biblioteca de plantillas genéricas, para las tareas más comunes, también mejoraría la reusabilidad del código.

Cuando programamos, muchas veces nos encontramos con un conjunto de tareas similares para resolver y que corresponden a diferentes aspectos de un mismo problema. Una forma de afrontar esta situación es escribir un procedimiento específico y eficiente para cada una de estas tareas. Podemos enumerar dos desventajas en esta forma de proceder:

1. Debe de realizarse un sobrexceso de programación, lo que aumenta el coste de creación del programa y de su verificación.

2. El mantenimiento del programa se hace más dificultoso, ya que un cambio en las especificaciones puede requerir el cambio de cada uno de los procedimientos.

Con ser grave la primera de las deficiencias apuntadas, la segunda es la que puede producir mayores costes a largo plazo. Muchos estudios indican que el mayor coste en el *ciclo de vida* de un programa no es el coste inicial de diseño, codificación y verificación, sino el coste posterior asociado al mantenimiento del programa mientras está en producción y uso. Una solución alternativa, que elimina las deficiencias comentadas anteriormente, consiste en escribir un procedimiento altamente parametrizado capaz de solucionar cada uno de los aspectos del problema. Nuevamente, podemos apuntar dos deficiencias:

1. La dificultad propia de programar un procedimiento genérico que cubra todas las alternativas de forma eficiente.

2. La ineficiencia inherente a este tipo de procedimientos, ya que un procedimiento altamente parametrizado gastará mucho de su tiempo de ejecución en la comprobación e interpretación de los parámetros y (relativamente) poco tiempo en los cómputos que debe realizar.

La EP puede ayudarnos a vencer esta disyuntiva. Podemos escribir un procedimiento genérico altamente parametrizado (posiblemente ineficiente) y utilizar un evaluador parcial para especializarlo, suministrando los valores de los parámetros adecuados para cada una de las tareas especificas a resolver. Esto permite obtener automáticamente un conjunto de procedimientos especificos y eficientes, tal y como deseábamos.

### 3.3. Autoaplicación y Generación Automática de Programas.

Uno de los objetivos más perseguidos en el campo de la evaluación parcial es lograr evaluadores parciales autoaplicables. La autoaplicación permite llevar a la práctica los resultados teóricos formulados por Futamura [25] y Ershov [24] (agrupados en la Proposición 2.2), que propician la generación automática de programas.

Dentro de la la generación automática de programas una de las aplicaciones más notables es la *generación de compiladores dirigida por la semántica* [54]; por ello entendemos lo siguiente: dada una especificación de un lenguaje de programación, basada en una semántica formal[4], transformarla automáticamente en un compilador. La motivación para la generación automática de compiladores es clara: el ahorro en esfuerzos de programación que supone la construcción de un compilador, que en ocasiones no es correcto con respecto a la semántica propuesta para el lenguaje que compila. La corrección de la EP permite que la transformación automática de una especificación semántica de un lenguaje en un compilador haga desaparecer estos errores. Las tareas de diseñar la especificación de un lenguaje, escribir el compilador y mostrar la corrección del compilador, se reducen a una sola tarea: escribir la especificación del lenguaje en una forma adecuada para ser la entrada de un generador de compiladores.

Como puede apreciarse, la EP y la autoaplicación tiene unas posibilidades muy prometedoras. Aunque todavía se necesita algún esfuerzo de investigación para entender perfectamente su teoría y sus técnicas prácticas, la EP ya ha tenido sus primeras aplicaciones industriales en diversos campos tecnológicos. También se ha aplicado a la resolución de problemas dentro del ámbito de la inteligencia artificial. En el próximo apartado se enumeran algunos ejemplos de todas estas aplicaciones.

## 4. Aplicaciones de la Evaluación Parcial.

La EP ha sido aplicada intensivamente tanto a los lenguajes imperativos (e.g., el lenguaje C) como a los declarativos (e.g., el lenguaje PROLOG, el lenguaje Scheme y otros) y extensivamente a una gran variedad de problemas concretos, una muestra de los cuales se presenta a continuación:

1. **Consultas en bases de datos.**

   Un evaluador parcial puede compilar una consulta determinada en un programa de búsqueda especializado, cuya tarea es responder a esa consulta específica. Si la pregunta con respecto a la que se especializa es genérica, el programa de búsqueda especializado servirá para responder a una clase de preguntas, compuesta por instancias de esa pregunta más general, de forma más eficiente [53]. Otros trabajos en esta misma línea son [18, 32].

2. **Demostración automática de teoremas.**

   El uso de la lógica y de sus sistemas deductivos no es la única via para lograr la demostración automática de teoremas. En un contexto de programación funcional, en [56, 59], Valentin Turchin muestra un método alternativo para la demostración automática de teoremas empleando técnicas de evalución parcial. En estos traba-

[4] Nótese que la semántica operacional de un lenguaje puede considerarse que es un intérprete (abstracto) del lenguaje

jos se emplea un evaluador parcial, allí denominado *supercompilador*, para realizar demostración de teoremas provenientes de la aritmética. Cuando el empleo del supercompilador sobre la expresión que se desea probar no conduce de manera directa a una prueba, se muestra que es posible alcanzarla aplicando el supercompilador al proceso de supercompilación mismo, realizando lo que Turchin denomina una transición a un metasistema (*metasystem transition*).

3. **Gráficos por computador.**

   El problema del *trazado de rayos* consiste en computar la trayectoria de un rayo de luz en un escenario (i.e., una colección de objetos tridimensionales). La escena es un dato que no cambia durante el cómputo de la trayectoria, lo que hace que éste sea un problema muy adecuado para su tratamiento mediante la técnica de la evaluación parcial. En [50], T. Mogensen ha optimizado el procedimiento de trazado de rayos, utilizado habitualmente en los programas para la construcción de gráficos por computador, especializándolo para una escena fijada. En los experimentos realizados por Mogensen, la versión especializa resultó entre 8 y 12 veces más rápida que el programa original.

4. **Mantenimiento del software y comprensión de programas.**

   En [14, 15] se desarrollan técnicas para el mantenimiento del software y la comprensión de programas (*program understanding*) basadas en la EP. Los autores desarrollan un evaluador parcial para programas escritos en el lenguaje FORTRAN que aplican a la comprensión de viejos programas científicos (para la gestión de centrales atómicas y satélites de comunicaciones). Con el paso del tiempo, estos programas se han hecho inmanejables, debido a numerosas extensiones, requiriéndose un gran esfuerzo para su mantenimiento. El evaluador parcial utiliza, principalmente, propagación de constantes y simplificación de alternativas por una de sus ramas con el fin de obtener un programa (especializado para ciertos valores de sus variables) más simple y fácil de entender.

5. **Planificación de tripulaciones en compañias aéreas.**

   En una compañía aérea, después de los gastos en combustible se sitúan en cuantía los gastos que originan el movimiento de las tripulaciones. En las grandes compañias aéreas estos gastos pueden superar el billón de dólares. Por consiguiente, se ha invertido un gran esfuerzo en mejorar la planificación relativa a estos aspectos. En [10] se presenta una solución a este problema basada en el uso de la EP para la mejora de los complicados programas de planificación que son necesarios. El sistema de planificación de tripulaciones desarrollado por Carmen Systems AB incorpora un evaluador parcial, escrito en el lenguaje funcional Haskell, que consigue aumentos en la eficiencia con respecto a los programas originales de entre un 30 % y 65 %. Si se tiene en cuenta que la ejecución de este tipo de programas requiere varias horas, ese aumento en la eficiencia supone una ganancia considerable. En la actualidad el sistema es utilizado por Lufthansa, la mayor compañía aérea de la Unión Europea.

6. **Protocolos de procedimiento de llamada remota.**

   La EP también se ha empleado para la optimización de *software* de sistemas. Un protocolo de procedimiento de llamada remota permite que un procedimiento remoto se ejecute aparentemente como si se tratase de uno local, siendo transparente para el usuario del sistema el hecho de que, en realidad, los cómputos tengan lugar en una máquina remota dentro de un sistema operativo distribuido. En [47] se informa de la optimización de partes del protocolo RPC (*Remote Procedure Call*) de Sun, mediante el empleo de Tempo [22], un evaluador parcial para el lenguaje C. Se han obtenido mejoras en la eficiencia del orden del 1.35 en los procedimientos de codificación/decodificación de argumentos y de hasta un 3.75 en el protocolo RPC mismo.

7. **Redes neuronales.**

   En general, el entrenamiento de una red neuronal emplea mucho tiempo de cómputo. En [35] se ha aplicado un evaluador parcial a un simulador para el entrenamiento de redes neuronales escrito en el lenguaje C. El programa especializado resultante transforma el programa original en un simulador más eficiente para una topología fijada de la red neuronal. Las mejoras en la eficiencia observadas oscilaron entre el 25 % y el 50 %, que pueden considerarse muy adecuadas debido a la gran cantidad de tiempo de cómputo que requiere el entrenamiento de una red neuronal.

8. **Simulación de circuitos.**

   Los simuladores de circuitos toman como entrada una descripción de un circuito eléctrico, construyen las ecuaciones diferenciales que describen su comportamiento y las resuelven mediante el empleo de métodos numéricos. Berlin y Weise [11] citan mejoras importantes en la eficiencia como resultado de especializar un simulador de circuitos, escrito en el lenguaje funcional Scheme, para un circuito particular.

9. **Software adaptable.**

   Recientemente, en [20] se ha estudiado cómo emplear la EP para incluir un comportamiento adaptable en programas Java existentes. Para conseguir este tipo de comportamiento se emplean las denominadas *clases especializadas*. La idea que se esconde detrás de las clases especializadas consiste en asociar implementaciones alternativas a una clase Java ordinaria. Las implementaciones alternativas se diferencian por su estado interno (una serie de predicados sobre los atributos). El comportamiento adaptable consiste en utilizar la implementación alternativa adecuada al estado interno del objeto. De esta forma, el objeto se adapta por sí mismo a la situación descrita por la clase especializada. Aqui, el papel de la evaluación parcial es suministrar, automáticamente, una implementación optimizada para cada caso específico.

   Las técnicas de EP y adaptabilidad del software también se han aplicado a los sistemas operativos [13].

10. **Verificación de programas.**

    Una de las mayores dificultades a la hora de emplear técnicas de verificación formal, como la comprobación de modelos (*model checking*), es la generación de modelos a partir del programa fuente que puedan constituir la entrada de diversas herramientas para la comprobación de modelos. Para cubrir esta tarea es necesario el empleo de técnicas muy sofisticadas de transformación, abstracción y análisis de programas [23]. En [33] se utiliza una combinación de las técnicas de interpretatación abstracta y EP para la construcción de modelos abstractos, a partir del código fuente de programas escritos en el lenguaje ADA, para la verificación de dichos programas. Posteriormente, esas mismas técnicas se han aplicado al lenguaje Java dentro del proyecto Bandera (ver más adelante en el próximo apartado).

# 5. La Evaluación Parcial en Internet.

Sin animo de ser exhaustivo, en este apartado se enumeran y comentan algunas direcciones URL de interés sobre el tema de la EP.

1. **Informaciones de caracter general**.

   • `http://www.dina.kvl.dk/~sestoft/pebook/`

   Información y material en línea relacionado con el libro, ya clásico, "Partial Evaluation and Automatic Program Generation" de N.D. Jones, C.K. Gomard y P. Sestoft [38].

   • `http://www.cis.ksu.edu/~hatcliff/pe-overview.html`

   Un breve resumen sobre el tema de la EP escrito por el propio N.D. Jones, uno de los máximos expertos del área.

2. **Proyectos y grupos de investigacion**.

   • Proyecto Bandera:

   `http://www.cis.ksu.edu/santos/bandera/`

   Proyecto del laboratorio SAnToS (*Laboratory for Specification, Analysis and Transformation of Software*) de la Universidad Estatal de Kansas. El objetivo del proyecto es la definición de técnicas avanzadas de procesamiento y transformación de lenguajes para la construcción de modelos que permitan la verificación de programas.

   • Grupo CLIP Lab:

`http://www.clip.dia.fi.upm.es/`

El grupo CLIP Lab (*Computational Logic, Implementation, and Parallelism Laboratory*) de la Universidad Politécnica de Madrid. Este grupo tiene una amplia experiencia en el campo de la transformación y optimización de programas declarativos mediante técnicas de especialización abstracta.

- Proyecto Compose:

`http://www.irisa.fr/compose/`

Proyecto conjunto del INRIA (*Institut National de Recherche en Informatique et en Automatique*), el IRISA (*Institut de Recherche en Informatique et Systèmes Alèatoires*) y la Universidad de Rennes para el diseño y desarrollo de programas y sistemas adaptables mediante el empleo de técnicas de EP.

- Grupo DTAI:

`http://www.cs.kuleuven.ac.be/~dtai/`

El grupo DTAI (*Declaratieve Talen en Artificiele Intelligentie*) de la Universidad Católica de Leuven tiene como temas principales de estudio los lenguajes declarativos, aprendizaje máquina y representación del conocimiento. También realizan una extensa investigación en el campo de la EP de programas lógicos.

- Grupo ELP:

`http://www.dsic.upv.es/users/elp/elp.html`

Grupo ELP (Extensiones de la Programación Lógica) de la Universidad Politécnica de Valencia. Sus áreas de interés se orientan, fundamentalmente, a la integración de lenguajes lógicos y funcionales y a la EP de programas lógico–funcionales, así como otras técnicas de transformación y manipulación de programas.

- Grupo TOPPS:

`http://www.diku.dk/research-groups/topps/`

El grupo TOPPS (*Teori Og Praksis for ProgrammeringsSprog*) del Departamento de Ciencias de la computación de la Universidad de Copenhague (DIKU - *Datalogisk Institut Københavns Universitet*) puede considerarse como el grupo pionero en el estudio de la EP. Con más de veinte años de experiencia, el grupo ha impulsado una buena parte de los desarrollos teóricos y prácticos (ver el subapartado de productos más adelante) en este área de conocimiento. El grupo sigue siendo uno de los más activos en las áreas de análisis de programas basado en la semántica y la manipulación de programas en general.

3. **Evaluadores parciales y otros productos**.

- C-Mix:

`http://www.diku.dk/research-groups/topps/activities/cmix/`

C-Mix es un evaluador parcial *offline* para el lenguaje C desarrollado por el grupo TOPPS del DIKU, Dinamarca [9]. C-Mix especializa los programas en tiempo de compilación. En la actualidad se está revisando su implementación e introduciendo nuevas mejoras.

- DPPD:

`http://www.staff.ecs.soton.ac.uk/~mal/systems/dppd.html`

DPPD es el acrónimo de "*Dozens of Problems for Partial Deduction library of benchmarks*", una librería de programas lógicos mantenida por M. Leuschel y que pretende llegar a ser un repositorio de programas estándar de prueba con los que medir la potencia de un evaluador parcial para programas lógicos. Estos programas de prueba pueden adartarse fácilmente a las peculiaridades sintácticas de otros lenguajes declarativos.

- ECCE:

`http://www.staff.ecs.soton.ac.uk/~mal/systems/ecce.html`

Un evaluador parcial *online* para programas lógicos implementado por M. Leuschel en la Universidad Católica de Leuven, Belgica [42]. La implementación de ECCE está basada, principalmente, en los siguientes trabajos de investigación [28, 44, 45] y [46].

- INDY:

`http://www.dsic.upv.es/users/elp/indy/`

El sistema INDY (*Integrated Narrowing - Driven specialization system* [2, 5]) es un evaluador parcial *online* para programas lógico–funcionales desarrollado por E. Albert y G. Vidal en la Universidad Politécnica de Valencia, España. INDY está implementado en SICStus Prolog v3.6 e incorpora las técnicas desarrollas en [1, 3, 6] y [8].

- JSCC:

`http://www.irisa.fr/compose/jscc/`

Un especializador de clases Java desarrollado, dentro del proyecto Compose, en la Universidad de Rennes, Francia [20].

- Mixtus:

`http://www.sics.se/ps/mixtus.html`

Mixtus es un evaluador parcial *online* para el lenguaje Prolog desarrollado por D. Sahlin en el SICS (*Swedish Institute of Computer Science*), Suecia. Mixtus [52] acepta objetivos instanciados parcialmente y todo tipo de predicados predefinidos de Prolog.

- SAGE:

`http://www.cogsci.ed.ac.uk/~corin/goedel.html`

SAGE (*Self-Applicable Goedel partial Evaluator* [31]) es un evaluador parcial autoaplicable para programas Goedel desarrollado por Corin Gurr en la Universidad de Bristol, Inglaterra. Debido a su capacidad para la autoaplicación se ha utilizado como herramienta para la metaprogramación y en la obtención de un generador de compiladores.

- Schism:

`http://www.irisa.fr/compose/schism/`

Schism es un evaluador parcial *offline* autoaplicable desarrollado por Charles Consel a finales de los ochenta, para un subconjunto puro (de primer orden y libre de efectos laterales) del lenguaje Scheme [19]. El sistema ha seguido desarrollándose y, en la actualidad, algunas de sus características principales son: *binding-time analysis* polivariante, admite la especialización de funciones de orden superior y estructuras de datos estáticas parciales.

- Similix:

`http://www.diku.dk/research-groups/topps/activities/similix.html`

Similix es un evaluador parcial *offline* autoaplicable para un subconjunto (de primer orden) del lenguaje Scheme. Fue implementado por Bondorf y Danvy en el DIKU [17] y Bondorf lo extendió posteriormente a un amplio subconjunto de orden superior de Scheme. Similix puede tratar estucturas de datos estáticas parciales y admite programas que contienen un uso limitado de efectos laterales (e.g., operaciones de entrada/salida).

- SML-mix:

`http://www.diku.dk/research-groups/topps/activities/sml-mix.html`

SML-mix es un prototipo de evaluador parcial para un subconjunto apreciable del lenguaje funcional Standard ML. Fue desarrollado primariamente por L. Birkedal y M. Welinder en el DIKU, Dinamarca [12].

- Tempo:

`http://www.irisa.fr/compose/tempo/`

Tempo es un evaluador parcial *offline* para el lenguaje C [22] que está siendo desarrollado conjuntamente por el IRISA, el INRIA y la Universidad de Rennes dentro del proyecto Compose. Tempo también puede especializar programas en tiempo de ejecución. Permite especialización modular de programas (i.e., especializar solamente parte de un programa). Tempo puede considerarse como el evaluador parcial para C más preciso de entre todos los existentes.

# 6. Discusión y Conclusiones.

Antes de concluir este artículo es conveniente discutir un aspecto teórico que puede haber quedado oculto en la exposición y que todavía requerirá de un esfuerzo de investigación considerable en el futuro hasta que sus posibilidades queden completamente claras. Dado un conjunto de sistemas $\{S_1, \ldots, S_n\}$ de un cierto tipo, supóngase que se unen en un nuevo sistema $S^*$ que contiene los sistemas $S_i$ como subsistemas y que incluye una serie de mecanismos adicionales que le permiten controlar, modificar y reproducir estos subsistemas. Al sistema $S^*$ se le denomina *metasistema* (con respecto a los $S_i$) y a la creación de $S^*$ se le denomina *transición a un metasistema*, en la nomenclatura introducida por Valentin Turchin [58, 59]. En sus trabajos, Turchin analiza el proceso del razonamiento humano como compuesto por tres actividades principales: i) Computación o deducción; ii) Generalización, abstracción o inducción y iii) Transición a metasistemas. Para resolver problemas, los humanos hacemos un uso intensivo de estos instrumentos en una determinada secuencia. Primero, una vez modelado y formalizado el problema, utilizamos sistemas de deducción y razonamiento deductivo para confirmar la validez de una afirmación y obtener respuestas (i.e., computar). A lo largo de este proceso, si en los resultados observamos la aparición de patrones, surge la posibilidad de generalizar. Si fallamos en la obtención de una solución a nuestro problema, analizamos el porqué examinando el proceso de aplicación de las reglas de deducción; i.e., efectuamos una transición a un metasistema del sistema deductivo. Este análisis puede dotarnos de reglas nuevas y más poderosas para la resolución del problema. Si volvemos a fallar, realizamos una nueva transición a un metasistema y analizamos nuestras técnicas de búsqueda de nuevas reglas. Naturalmente, las transiciones podrían proseguir hasta el infinito si nos encontramos en presencia de un problema insoluble. El proyecto de Supercompilación [55, 56, 57, 58, 59, 60], además de suministrar herramientas para implementar las dos primeras actividades del razonamiento humano, ensaya la posibilidad de efectuar múltiples transiciones a metasistemas mediante la aplicación reiterada de la supercompilación a un problema concreto. Así pues, la capacidad de autoaplicación de los evaluadores parciales puede constituir una herramienta poderosa para la resolución de problemas. Turchin denomina *metacomputación* a la combinación de las técnicas de transición a metasistemas con la evaluación parcial [58, 59]. Otros autores que han estudiado la transición a metasistemas, buscando su formalización, son [27, 29, 30, 34].

En este artículo se ha introducido la técnica de transformación automática de programas denominada evaluación parcial y se han descrito sus características y principales objetivos. También se han presentado algunas de sus aplicaciones prácticas y otras que son de interés para todo aquél que desarrolla su tarea investigadora en el campo de la inteligencia artificial, como son: la demostración automática de teoremas; la resolución de problemas; la especialización de redes neuronales y la obtención de software adaptable. Por la relevancia de los objetivos perseguidos, su incidencia positiva en el desarrollo del *software* y el interés de sus aplicaciones, se puede afirmar que las ventajas prácticas de la evaluación parcial están fuera de toda duda.

# Reconocimientos

Agradezco a María Alpuente la lectura atenta de un primer borrador de este trabajo y sus valiosos comentarios que han contribuido grandemente a su mejora.

# Referencias


[1] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In *Proc. of SAS*, Springer LNCS 1503:262–277, 1998.

[2] E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. TR DSIC-II/12/98, UPV, 1998. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

[3] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM*, *ACM Sigplan Notices* 32(12):151–162, 1997.

[4] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A transformation system for lazy functional logic programs. In *Proc. of FLOPS*, Springer LNCS 1722:147–162, 1999.

[5] M. Alpuente, M. Falaschi, and G. Vidal. Experiments with the Call-by-Value Partial Evaluator. DSIC-II/13/98, UPV, 1998. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

[6] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.

[7] M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.

[8] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of inductively sequential

functional logic programs. In *Proc. of ICFP*, pp 273–283, ACM Press, 1999.

[9] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU TR 94/19, U. of Copenhagen, 1994.

[10] L. Augustson. Partial Evaluation in Aircraft Crew Planning. In *Proc. of PEPM, ACM Sigplan Notices*, 32(12):127–136, 1997.

[11] A. Berlin and D. Weise. Compiling Scientific Code Using Partial Evaluation. *IEEE Computer*, 23(12):25–37, 1990.

[12] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's Thesis, DIKU TR 93/22, U. of Copenhagen, 1993.

[13] A. Black, C. Consel, C. Cowan, C. Krasik, C. Pu, E.N. Volanschi, and J. Walpole. Specialization classes: An object framework for specialization. In *fifth IEEE IWOOOS*, pp 286–300. 1996. Also available at `http://www.irisa.fr/compose/papers/#spec_decl`.

[14] S. Blazy and P. Facon. Partial evaluation for the understanding of fortran programs. In *Soft. Engineering and Knowledge Engineering*, pp 517–525, 1993.

[15] S. Blazy and P. Facon. An automatic interprocedural analysis for the understanding of scientific application programs. In *Dagstuhl Int'l Seminar on Partial Evaluation*, Springer LNCS 1110:1–16, 1996.

[16] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[17] A. Bondorf. *Similix Manual, System Version 4.0.* TR 94/19, DIKU, U. of Copenhagen, 1994.

[18] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. In *Proc. of first Int'l Conf. on Deductive and Object-Oriented Databases*, pp 25-44. Elsevie, North-Holland, 1990.

[19] C. Consel. New Insights into Partial Evaluation: The Schism Experiment. In *Proc. of ESOP*, Springer-Verlag LNCS 300:236–246, 1988.

[20] C. Consel, C. Cowan, G. Muller, and E.N. Volanschi. Declarative specialization of object-oriented programs. In *ACM SIGPLAN conf. on OOPSLA*, pp 286–300, 1997. Also available as TR RR-3118, INRIA, 1997; at `http://www.irisa.fr/compose/papers/#spec_decl`.

[21] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of POPL*, pp 493–501, ACM Press, 1993. Also available at `http://www.irisa.fr/compose/pe/pe_resources.html#references`.

[22] C. Consel, L. Hornof, F.Ñoël, J.Ñoyé, and E.N. Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialisation. In *Proc. of Dagstuhl Seminar on Partial Evaluation*, Springer LNCS 1110:54–72, 1996.

[23] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from java source code. In *Proc. of ICSE 2000* (to appear). Available at `http://www.cis.ksu.edu/santos/bandera/`.

[24] A.P. Ershov. On the Partial Compitation Principle. *Information Processing Letters*, 6(2):38–41, 1977.

[25] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[26] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM*, pp 88–98. ACM Press, 1993.

[27] R. Glück. Towards Multiple Self-Application. In *Proc. of PEPM*, ACM Sigplan Notices, 26(9):309–320, 1991.

[28] R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In *Proc. of PLILP*, Springer LNCS 1140:152–166, 1996.

[29] R. Glück and A.V. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In *Proc. of WAS*, Springer LNCS 724:112–123, 1993.

[30] R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, Springer LNCS 1110:137–160, 1996.

[31] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel.* PhD thesis, Dep. of Computer Science, U. of Bristol, 1994.

[32] J. Han. Chain-SPlit Evaluation in Deductive Databases. In *Proc. of Eighth International Conference on Data Engineering, Tempe, AZ, USA*, pp 376-384. IEEE Computer Society, New York, 1992.

[33] J. Hatcliff, M.B. Dwyer, and S. Laubach. Staging static analyses using abstraction-based program specialization. In *Proc. of PLILP.* Springer LNCS 1490, 1998.

[34] J. Hatcliff and R. Glück. Reasoning about hierarchies of online program specialization systems. In *Partial Evaluation, Int'l Seminar,*

*Dagstuhl Castle, Germany*, Springer LNCS 1110:161–182, 1996.

[35] H.F. Jacobsen. Speeding up the Back-Propagation Algorithm by Partial Evaluation to Ray-Tracing. TR, Project 90-10-13, DIKU, U. of Copenhagen, 1990.

[36] N.D. Jones. Automatic Program Specialization: A Re-Examination from Basic Principles. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pp 225–282. North-Holland, 1988.

[37] N.D. Jones. What *not* to Do When Writing an Interpreter for Specialisation. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, Springer LNCS 1110:216–237, 1996.

[38] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[39] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[40] P. Julián-Iranzo. *Especialización de Programas Lógico–Funcionales Peresosos.* PhD thesis, DSIC-UPV, May. 2000.

[41] S.C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, NJ, 1952.

[42] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Technical report, 1998. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[43] M. Leuschel. *On the Power of Homeomorphic Embedding for Online Termination.* In *Proc. of SAS*, Springer LNCS 1503:230-245, 1998. Also available as Technical Report DSSE-TR-98 at `http://www.staff.ecs.soton.ac.uk/ mal/ mypublications.html#DSSE9811.abstract`.

[44] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In *Proc. of JICSLP*, pp 319–332. The MIT Press, 1996.

[45] M. Leuschel and B. Martens. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. TR CW-220, Dep. of Computer Science, K.U. Leuven, 1995.

[46] M. Leuschel and M.H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proceedings of LOPSTR*, Springer LNCS, 1996.

[47] R. Marlet, G. Muller, and E.N. Volanschi. Scaling up Partial Evaluation for optimizing the sun commercial rpc protocol. In *Proc. of PEPM*, *ACM Sigplan Notices*, 32(12):116–126, 1997.

[48] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. CSTR-94-16, Computer Science Dep., U. of Bristol, 1994.

[49] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. of ICLP*, pp 597–611. MIT Press, 1995.

[50] T. Mogensen. The Application of Partial Evaluation to Ray-Tracing. Master's Thesis, DIKU, U. of Copenhagen, 1986.

[51] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[52] D. Sahlin. The Mixtus Approach to Automatic Partial Evaluation of Full Prolog. In *Proc. of North American Conf. on Logic Programming*, pp 377–398. The MIT Press, 1990.

[53] C. Sakama and H. Itoh. Partial Evaluation of Queries in Deductive Databases. *New Generation Computing*, 6(2&3):249-258, 1988.

[54] M. Tofte. *Compiler Generators: What They Can Do, What They Might Do, and What They Will Probably Never Do. EATCS Monographs* 19. Springer-Verlag, 1990.

[55] V.F. Turchin. The language Refal, the Theory of Compilation and Metasystem Analysis. CCSR-20, Courant Institute of Mathematical Sciences, New York University, 1980.

[56] V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. In *Proc of ALP*, Springer LNCS 85:645–657, 1980.

[57] V.F. Turchin. The Concept of a Supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.

[58] V.F. Turchin. Program Transformation with Metasystem Transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[59] V.F. Turchin. Metacomputation: Metasystem Transitions plus Supercompilation. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, Springer LNCS 1110:481–509, 1996.

[60] V.F. Turchin and A.P. Nemytykh. A Self-applicable Supercompiler. TR-95-010, Computer Science Dep., The City College of New York, 1995.