PASADO, PRESENTE Y FUTURO DE LOS LENGUAJES DE PROGRAMACIÓN

Ricardo Peña Marí
Catedrático de Lenguajes y Sistemas Informáticos

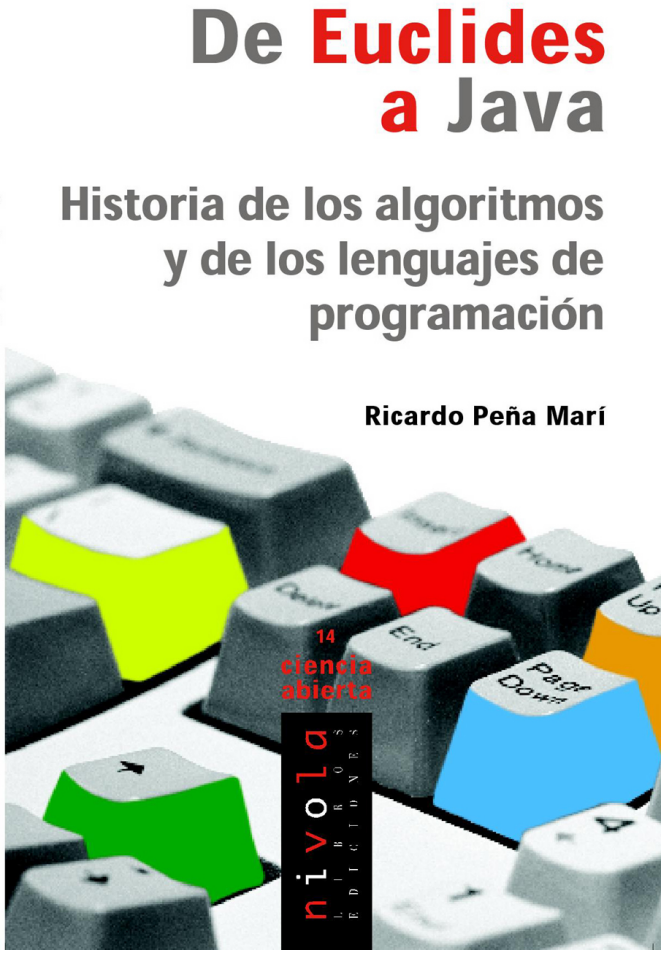
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Conferencias de Posgrado, Facultad de Informática UCM, junio de 2017

Índice

Índice

El libro De Euclides a Java



FORTRAN: John Backus, 1954-1957

```
C
C PROGRAMA QUE CALCULA EL MAXIMO COMUN DIVISOR DE DOS NUMEROS
C
      INTEGER X, Y, A, B
      READ (1,80) X, Y
      A = X
      B = Y
10    IF (A-B) 20, 30, 40
20    A = A - B
      GO TO 10
40    B = B - A
      GO TO 10
30    WRITE (2,90) A
80    FORMAT (2I5)
90    FORMAT (12H RESULTADO=,I5)
      END
```

John Backus

LISP: John McCarthy, 1957-1959

```
(DEFUN MAPLIST (F,XS)
       (COND (NULL XS)
             NIL
             (CONS (F (CAR XS))
                   (MAPLIST F (CDR XS))
             )
       )
)
```

Programa que aplica una función F a cada elemento de una lista XS

John McCarthy

Primer lenguaje en introducir:

- Recursión
- Orden superior
- Recolector de basura

La crisis (1968)

- Durante los 60 se desarrollan gran cantidad de lenguajes de alto nivel: ALGOL-60, COBOL, JOVIAL, MAD, NELLIAC, Algol-W, BASIC, GPSS, SNOBOL,...
- La potencia del hardware también crece exponencialmente gracias al desarrollo en esos años de los primeros circuitos integrados
- Aparición de la interrupción y, como consecuencia, de la multiprogramación
- Época de enormes desarrollos y enormes desastres. El punto de inflexión lo marca el fracaso del sistema operativo OS/360 de IBM (5.000 personas-año)
- El límite de los sistemas razonablemente fiables se sitúa alrededor de las 100.000 líneas de código
- Los lenguajes PL/I y Algol 68 contribuyen a la crisis
- Reconocimiento oficial de lo inapropiado de la tecnología software: Conferencia de Garmisch (1968) sobre Ingeniería de la Programación

Hitos posteriores más relevantes

1970-74 Programación estructurada, Pascal (N. Wirth)

1972- Programación lógica, Prolog (A. Comerauer)

1968-74 Semáforos, monitores, (E.W. Dijkstra, C.A.R. Hoare)

1972-80 Tipos abstractos de datos, CLU, Modula-2, Ada

1978-85 Procesos secuenciales comunicantes, Occam (INMOS Ltd.)

1980-83 Orientación a objetos, Smalltalk (A. Kay), C++ (B. Stroustrup)

1978-90 Programación funcional, Hope, SML, Miranda, Haskell

1993- Paralelismo de datos, High Performance Fortran

1995- Navegadores de Internet, Java

Enumeración de "aspectos"

- Es difícil clasificar la ingente cantidad de lenguajes desarrollados desde 1960.
- Ciertos aspectos aparecen por primera vez en alguno de ellos, pero luego son incorporados a lenguajes posteriores.
- Es preferible hablar de aspectos de interés, y de su evolución, e indicar para cada lenguaje cuáles de dichos aspectos incorpora.
- El aspecto más relevante es el modelo de cómputo:
  - paradigma imperativo
  - paradigma lógico
  - paradigma funcional
- Otros aspectos relevantes:
  - encapsulamiento y modularidad
  - concurrencia y paralelismo
  - sistemas de tipos
  - resolución de restricciones
  - metaprogramación
  - generación dinámica de HTML

Índice

El paradigma imperativo

- Sus lenguajes representan una abstracción de la máquina hardware sudyacente.
- El cómputo se concibe como una transformación incremental del estado.
- La instrucción estrella es la de asignación, de la forma $x := exp$, donde x es una variable, posiblemente subindicada.
- Esta instrucción se corresponde con el trasiego de datos entre la memoria y la unidad de proceso.
- El resto de las instrucciones se dedican a indicar el orden preciso en que se realizan estas asignaciones y a variar los subíndices.
- De esta forma, el cómputo avanza modificando incrementalmente el estado hasta alcanzar el estado final deseado.

El paradigma lógico (1)

```
1)  descendiente(X,Y) :- hijo(X,Y).
2)  descendiente(X,Y) :- hijo(Z,Y), descendiente(X,Z).
3)  hermano(X,Y) :- hijo(X,Z), hijo(Y,Z), X /= Y.
4)  sobrino(X,Y) :- hermano(Y,Z), descendiente(X,Z).
5)  hijo(miguel,pedro).
6)  hijo(sonia,pedro).
7)  hijo(sonia,juana).
8)  hijo(laura,juana).
9)  hijo(luis, sonia).
10) hijo(antonio,sonia).
11) hijo(olga,luis).
12) hijo(diana,antonio).
```

El paradigma lógico (2)

- El programa no especifica el orden de ejecución, sino que define un conjunto de propiedades y hechos.
- La repetición de cómputos está implícita en la recursión.
- Los condicionales están implíctos en la elección de cláusula mediante un ajuste de patrones especial llamado unificación.
- Varias respuestas ante una misma pregunta (no determinismo):

```
hermano(sonia,X)?  ->  X = miguel; X = laura
```

- Ejecución reversible:

```
descendiente(X,pedro)?  versus  descendiente(diana,Y)?
```

- La corrección de cada cláusula se puede inspeccionar independientemente
- El cómputo avanza mediante pasos de resolución, que pueden entenderse como una forma de deducción.
- Reducción de un orden de magnitud en el número de líneas de código fuente frente al paradigma imperativo.

El paradigma funcional (1)

```
data Ord a => Arbus a = Vacio | Unir (Arbus a) a (Arbus a)

insertar :: Ord a => a -> Arbus a -> Arbus a
insertar x Vacio = Unir Vacio x Vacio
insertar x (Unir i y d)
    | x >  y = Unir i y (insertar x d)
    | x == y = Unir i y d
    | x <  y = Unir (insertar x i) y d

crear :: Ord A => [a] -> Arbus a
crear = foldr insertar Vacio

inorden :: Ord a => Arbus a -> [a]
inorden Vacio        = []
inorden (Unir i x d) = inorden i ++ [x] ++ inorden d
```

El paradigma funcional (2)

- Ausencia de estado mutable: Ejecutar $\equiv$ Simplificar una expresión
- El paso elemental de ejecución es la β-reducción, consistente en reemplazar una parte izquierda por la correspondiente parte derecha de una definición.
- No hay control de secuencia, salvo la implícita en las dependencias de datos.
- El recorrido recursivo de las estructuras de datos se encapsula en funciones polimórficas de orden superior.
- Con ello se dispone de un amplio catálogo de bloques genéricos reutilizables:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- Introdujeron el sistema polimórfico de tipos y la inferencia automática.

El paradigma funcional (3)

- En realidad, el paradigma funcional aporta dos modelos de cómputo:
  - La evaluación impaciente: corresponde al paso de parámetros por valor.
  - La evaluación perezosa: termina más a menudo que la impaciente, no realiza cálculos innecesarios, y admite objetos infinitos:

```
nats = 0 : map (+1) nats
```

- Otra aportación es el orden superior: las funciones son tratadas como valores a todos los efectos. Pueden ser parámetros o/y resultados de otras funciones, almacenadas en estructuras de datos, y definidas dinámicamente.
- Su uso disminuye el esfuerzo de programación y la introducción de errores.
- Programar se parece más a componer piezas que a crear código nuevo.
- El paradigma imperativo soporta una versión más limitada del orden superior y la presencia de posibles efectos laterales limita aún más dicho uso.

Índice

Abstracciones y facilidades de ocultamiento

- Los primeros LP incluían tan solo procedimientos y funciones con parámetros como único mecanismo de abstracción.
- Ello permitió un cierto grado de encapsulamiento ya que es posible utilizar una función sin conocer cómo está implementada.
- Pero es insuficiente para basar en ella la descomposición modular de un programa grande.
- La abstracción llegó algo mas tarde a las representaciones de datos con la aparición del concepto de tipo abstracto de datos.
- Dicho concepto, junto con la creación dinámica de objetos, conteniendo cada uno un valor del tipo abstracto, dio lugar a la idea de clase de la programación orientada objetos.
- Basados en estas ideas, la mayoría de los lenguajes incluyen actualmente alguna noción de módulo que permita exportar al entorno circundante cierta información, y a la vez que se oculta otra.

Los tipos abstractos de datos

- Concepto acuñado hacia 1972. Aportaciones:
  1. Una noción de ocultamiento de la representación de un tipo
  2. Una interfaz visible: conjunto de operaciones que operan con valores del tipo
  3. Una técnica de especificación formal del comportamiento observable
  4. Un concepto de genericidad: tipos pasados como parámetros a un tipo.
- Da lugar a un nuevo concepto de módulo y a nuevos ámbitos de visibilidad de los identificadores.
- Produce herramientas de prototipado y de demostración de propiedades basadas en técnicas de reescritura.
- Lenguajes pioneros: CLU (B. Liskov, 1974), Modula (N. Wirth, 1975), Ada (DoD, 1975-1980).
- También son uno de los orígenes de los lenguajes orientados a objetos: Smalltalk (A. Kay, 1980), C++ (B. Stroustrup, 1983), Java (J. Gosling, 1995)

La programación orientada a objetos (1)

- El segundo origen de la POO es la noción de herencia introducida por primera vez en un derivado de Algol-60, Simula 67 (Nygaard y Dahl, 1967)
- El paradigma, tal como hoy lo conocemos, se debe a la visión de Alan Kay (*"Todo son objetos"*) y a su lenguaje Smalltalk:
  1. Lenguaje compilado a una máquina virtual
  2. Creación dinámica de objetos, recolector de basura
  3. Conceptos de clase, subclase, vinculación dinámica, clase abstracta,...
  4. Entorno gráfico interactivo, el primero basado en ventanas e iconos
  5. Amplia librería de clases con estructura jerárquica
- La herencia es un potente factor de reutilización: la subclase solo necesita definir los campos y métodos adicionales a los de la superclase .
- C++ añade la eficiencia necesaria para que el paradigma se difunda en la industria.
- Java añade la independencia de la máquina, el código móvil y el maridaje con los navegadores de Internet.

La programación orientada a objetos (2)

```
      // Clase para manipular fechas
public class Fecha {
      // Construye una fecha por defecto
  public Fecha ( )
     { dia = 1; mes = 1; año = 2000 }
      // Construye una fecha concreta
  public Fecha (int d, int m, int a)
     { dia = d; mes = m; año = a }
  public boolean MayorQue (Fecha f2)
     { return año >  f2.año ||
              año == f2.año &&
                      (mes >  f2.mes ||
                       mes == f2.mes && dia > f2.dia);
     }
      // Atributos privados de la clase
  private int dia;
  private int mes;
  private int año;
}
```

Una clase escrita en Java

Índice

Concurrencia (1)

- La disparidad entre la velocidad de la unidad de proceso y la de los periféricos condujo en los años 1960 a la invención de la interrupción, y con ella al paradigma de programación concurrente.
- La multiprogramación (i.e. los procesos concurrentes en una sola máquina) trajo consigo nuevos tipos de errores:
  - corrupción de los datos accedidos concurrentemente
  - errores dependientes del tiempo
  - interbloqueos
  - inanición
- Se tardaron muchos años en comprender el fenómeno. Fueron determinantes los trabajos de E.W. Dijkstra, y C.A.R. Hoare (1965-1975).
- Los primeros mecanismos, semáforos, regiones críticas, monitores, presuponen la existencia de una memoria común.
- Los siguientes, mensajes asíncronos, mensajes síncronos, llamada remota o rendezvous, pueden aplicarse tanto a memoria compartida como distribuida.

Concurrencia (2)

- La concurrencia se considera hoy el modo natural de organizar las tareas de los sistemas operativos, y de los sistemas que controlan procesos de la realidad (centrales telefónicas, servidores web, plantas industriales, etc.).
- El primer lenguaje de amplio uso en incorporar mecanismos concurrentes fue Ada (1980).
- Actualmente todos los LP imperativos incorporan alguna noción de hebra concurrente y mecanismos para sincronizarlas.
- Los lenguajes funcionales y lógicos también tienen versiones concurrentes: Concurrent Haskell, Concurrent ML, Prolog Concurrente, P-Prolog.
- Algunos de ellos, como Erlang, han nacido directamente con la pretensión de programar sistemas distribuidos con miles de procesos y docenas de máquinas para albergarlos.

Paralelismo (1)

- La programación paralela tiene como objetivo explotar al máximo la capacidad de cálculo de un conjunto de procesadores.
- Se utiliza para rebajar el tiempo de algoritmos intensivos en cómputo: predicción meteorológica, plegado de proteinas, factorización de números ...
- Los objetivos principales son repartir la carga del modo más uniforme posible, y disminuir el proceso dedicado a comunicaciones.
- Hay dos modelos básicos de cómputo, el paralelismo de tareas y el paralelismo de datos.
- En el primero, el programador crea una estructura de procesos (p.e. esquemas divide y vencerás, en tubería, de trabajadores replicados, etc.) y no todas las tareas realizan el mismo trabajo.
- En el segundo, el programador es responsable de distribuir los datos entre los procesadores, pero existe un solo código secuencial que es ejecutado síncronamente por todos ellos.

Paralelismo (2)

- En el modelo de paralelismo de datos el LP estándar es High Performance Fortran (1993), una variante de Fortran 90 con directivas para la distribución.
- En el paradigma funcional, Data Parallel Haskell (2007), también implementa dicho modelo.
- En el modelo de paralelismo de tareas hay dos estándares de librerías de paso de mensajes: PVM (Parallel Virtual Machine, 1990) y MPI (Message Passing Interface, 1993) .
- El paralelismo, además de un paradigma con lenguajes propios, también es un mecanismo de implementación de los lenguajes, especialmente en el ámbito declarativo (Eden, Glasgow Parallel Haskell, Parallel Prolog).
- En estos lenguajes, el programador no necesita especificar el orden de las acciones. El compilador y el sistema de soporte organizan la ejecución en paralelo de partes del programa que no interfieran entre sí. Las oportunidades son muchas debido a la ausencia de efectos laterales.

Índice

Evolución de los sistemas de tipos

- Los tipos de los primeros lenguajes se correspondían casi exactamente con los soportados por el hardware.
- Más adelante, se formaron dos corrientes: los que defendían una disciplina de tipos, en la que toda variable se declara con un tipo y el compilador fuerza un uso compatible con el mismo, y los que defendían un lenguaje sin tipos en tiempo de compilación.
- A la primera pertenecen por ejemplo Algol-60, Pascal, Ada y Java.
- A la segunda, Lisp, Prolog y Erlang.
- Los defensores de la primera alegan seguridad, los de la segunda flexibilidad.
- Los lenguajes funcionales (SML, 1980) encontraron el sistema de tipos con polimorfismo paramétrico, que parece reunir lo mejor de ambos mundos.
- Por un lado, no es necesario declarar el tipo de las variables. Por otro, una variable o una función pueden tener muchos tipos, lo que va a favor de la flexibilidad.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

El polimorfismo de subtipos

- Los lenguajes orientados a objetos soportan el llamado polimorfismo de subtipos, según el cual un objeto de una subclase es admisible en cualquier parte del texto donde sea admisible uno de la superclase.
- También suministran un tipo universal `Object` del cual toda clase es subtipo.
- Con ello es posible tener las ventajas del polimorfismo paramétrico, y otras adicionales como crear una lista con objetos de tipos distintos. En ese caso, a costa de la seguridad, o de introducir comprobaciones dinámicas.
- También soportan una forma de tipado en tiempo de ejecución, la vinculación dinámica, que escoge el método aplicable a un objeto según el tipo de este.
- Ello obliga a tener una representación en ejecución del tipo de un objeto.

La sobrecarga

- Se llama sobrecarga al hecho de poder emplear el mismo nombre para operaciones distintas de un LP. Algunas operaciones tales como $+$, $-$, $*$, $\leq$, están sobrecargadas en la mayoría de los lenguajes.
- A partir de Ada, la sobrecarga se permite también para las operaciones definidas por el usuario.
- Los lenguajes orientados a objetos, como C++ y Java, también la permiten. Un uso frecuente es definir distintos constructores de un objeto, todos ellos con el mismo nombre.
- El compilador determina la función apropiada en base al número y tipo de los argumentos.
- Haskell introduce las llamadas clases de tipos para definir operaciones sobrecargadas:

```
quicksort :: Ord a => [a] -> [a]
```

- Las clases abstractas y las interfaces de Java, proporcionan facilidades muy similares.

La genericidad

- Es una consecuencia directa de la teoría de tipos abstractos de datos (TAD) desarrollada durante los años 1970-80.
- Un TAD puede estar parametrizado por otros TADs anónimos de los que solo se conocen algunas de sus operaciones. Ejemplo: árbol binario de búsqueda de elementos provistos de una relación de orden.
- El primer LP que soportó la genericidad fue Ada: permite la declaración de un **package**, **procedure** o **function** genéricos.
- Pueden recibir como parámetros tipos, operaciones y constantes. Para tener construcciones ejecutables, ha de concretarse la construcción genérica con parámetros reales que reemplacen a los formales.
- La versión de C++ de 1990 incluyó las *templates*, clases que pueden recibir como parámetros tipos, otras clases, o constantes.
- Las versiones más modernas de Java también incluyen esta facilidad.

Otra forma de genericidad: el politipismo

- Permite definir operaciones aplicables a todos los tipos de datos, actuales o futuros. La definición se hace por inducción sobre la estructura del tipo.
- De momento, solo lo soportan los lenguajes funcionales:

```
class Sized a where
  size ::  a -> Int
  size a = gsize (from a)
class GSized f where
  gsize :: f a -> Int
instance GSized U1 where
  gsize U1 = 0
instance (GSized a, GSized b) => GSized (a :*: b) where
  gsize (x :*: y) = gsize x + gsize y
instance (GSized a, GSized b) => GSized (a :+: b) where
  gsize (L1 x) = gsize x
  gsize (R1 x) = gsize x
instance (GSized f) => GSized (M1 i c f) where
  gsize (M1 x) = gsize x
instance Sized a => GSized (K1 i a) where
  gsize (K1 x) = size x
instance Sized Int where
  size x = 1
instance Sized Char where
  size x = 1
```

Resumen de las aportaciones de los tipos

- Los tipos estáticos comenzaron siendo un factor de seguridad: el tipo se analiza durante la compilación y ello asegura que en ejecución se preservará.
- Forzar tipos correctos en tiempo de compilación:
  1. Detecta errores o despistes con una ínfima inversión de esfuerzo
  2. Asegura que no se producirán errores de tipos en ninguna ejecución
- Compárese con analizar los tipos en ejecución y con programar sin tipos.
- Las diversas formas de polimorfismo, la sobrecarga, la genericidad y el politipismo los convierte además en un potente factor de reutilización:
  - Cuanto más general sea el tipo de un componente, en más contextos puede ser aceptado como válido, y por tanto puede ser más reutilizado
- Programar sin tipos debería reservarse solo para resolver situaciones excepcionales y solo para programadores expertos.

Índice

La programación con restricciones

- Existen lenguajes especializados para expresar y resolver problemas con restricciones desde los años 1970. Son lenguajes de propósito específico.
- Un problema de este tipo consta de los siguientes elementos:
  - CSP $\equiv (V, D, C)$ COP $\equiv (V, D, C, F)$.
    - Variables: $V \equiv \{v_1, v_2, \ldots, v_n\}$
    - Dominios: $D \equiv \{d_1, d_2, \ldots, d_n\}$
    - Restricciones: $C \equiv \{c_1, c_2, \ldots, c_m\}$
  - Candidatos: combinaciones que asignan a cada v_i un valor de d_i.
  - Resolver: consiste en encontrar candidatos que satisfagan C.
  - Optimizar: consiste además en minimizar/maximizar una función objetivo F.
- Desde cualquier LP se puede utilizar una librería CSP/COP, pero el paradigma lógico ha integrado la resolución de restricciones de forma bastante natural.
- A partir de 1987 aparecen los lenguajes lógicos con restricciones (Constraint Logic Programming, o CLP). Todas las versiones actuales de Prolog incluyen el uso de las mismas.
- Tipos de restricciones: dominios finitos, igualdad, $\leq$, lineales sobre reales, racionales o enteros, cuadráticas, ...

La metaprogramación

- También llamada reflexividad (*reflection*), es la posibilidad de condicionar la compilación desde el propio del programa en compilación.
- Los primeros antecedentes son el sistema de macros de LISP, y los preprocesadores de C para realizar por ejemplo compilación condicional.
- Actualmente, todos los LP incluyen directivas de compilación o preprocesadores para especializar la compilación. Con ellos es posible incluso definir lenguajes empotrados dentro del lenguaje huesped.
- El lenguaje de plantillas de C++ es tan potente que permite definir cualquier función computable (p.e. el factorial) para ser ejecutada durante la compilación.
- En MetaML es posible escribir metacódigo que generará código en tiempo de ejecución. Dicho código estará, no obstante, bien tipado.
- Template Haskell permite inspeccionar los tipos de los módulos ya compilados, y generar código especializado para dichos tipos:

```
data T a = Tip a | Fork (T a) (T a)
reify :: Name -> Q Info
info = reify "T" ...
```

Los lenguajes de *script*

- La aparición de Internet y la necesidad de un creciente trasiego de páginas HTML y XML entre servidores y clientes, desató toda una pléyade de nuevas tecnologías y lenguajes.
- Casi desde el principio hubo un deseo de dotar a dichas páginas de animación e interactividad.
- Muy pronto surgieron los llamados lenguajes de *script*, lenguajes todo-terreno cuyo objetivo inicial era crear dinámicamente páginas HTML.
- Se les dotó de acceso a los mandatos del sistema operativo y a las bases de datos subyacentes con el fin de manipular ficheros, ejecutar programas, y obtener o modificar información.
- Aunque podríamos considerar de propósito especifico estos lenguajes, de hecho engloban en su seno lenguajes de programación completos y se escriben en ellos aplicaciones de creciente complejidad.
- Ejemplos: JavaScript, JSP, PHP, Ruby, Python (via API), ...

Índice

El presente: la confusión de las lenguas

- El momento actual se caracteriza por la combinación de varios paradigmas en un mismo lenguaje:
  - Programación lógico-funcional (Curry, TOY)
  - Programación funcional-concurrente (Erlang, Concurrent Haskell)
  - Programación funcional-paralela (GPH, Eden, Data Parallel Haskell)
  - Programación lógico-funcional-concurrente-orientada a objetos (Oz)
- A la vez, los lenguajes imperativos orientados objetos incorporan un creciente número de construcciones funcionales:
  - JavaSript: maps, folds, closures
  - Java 8: maps, folds, lambda abstractions
  - Ruby: lambda abstractions, closures, first-class continuations
  - Python: map, reduce, filter, list comprehensions, lambda abstractions
- Y algunos lenguajes funcionales incorporan facilidades para la orientación a objetos: Scala, F#
- ¿Estamos combinando lo mejor de ambos mundos?

El presente: la confusión de las lenguas

- El momento actual se caracteriza por la combinación de varios paradigmas en un mismo lenguaje:
  - Programación lógico-funcional (Curry, TOY)
  - Programación funcional-concurrente (Erlang, Concurrent Haskell)
  - Programación funcional-paralela (GPH, Eden, Data Parallel Haskell)
  - Programación lógico-funcional-concurrente-orientada a objetos (Oz)
- A la vez, los lenguajes imperativos orientados objetos incorporan un creciente número de construcciones funcionales:
  - JavaSript: maps, folds, closures
  - Java 8: maps, folds, lambda abstractions
  - Ruby: lambda abstractions, closures, first-class continuations
  - Python: map, reduce, filter, list comprehensions, lambda abstractions
- Y algunos lenguajes funcionales incorporan facilidades para la orientación a objetos: Scala, F#
- ¿Estamos combinando lo peor de ambos mundos?

El presente: el abandono del tipado estático

- La mayor parte de los lenguajes de *script* son interpretados y tienen tipado dinámico: Ruby, PHP, Python, JavaScript, ...
- Ello presenta dos inconvenientes:
  - Necesitan mas memoria y más tiempo, para poder comprobar los tipos en ejecución.
  - Solo se detectan los errores de tipos en los caminos realmente ejecutados.
- Además, la mayoría de ellos convierten automáticamente unos tipos en otros con excesiva flexibilidad (p.e. en Phyton las variables adquieren tipo al ser asignadas, en JavaScript, "5" * 5 devuelve 25, pero "a" * 5 es un error).
- Lo cual, unido a la mezcla de paradigmas, y a las amplias facilidades de metaprogramación que soportan, hace que estos lenguajes sean muy peligrosos de usar por programadores no expertos.
- ¿Vamos hacia una nueva crisis del software?

El futuro (1)

- Resumiendo, la evolución de los LP ha ido en las siguientes direcciones:
  1. Un creciente nivel de abstracción: tipos definidos por el usuario, tipos abstractos, clases, funciones de orden superior, construcciones para la concurrencia, etc.
  2. Mecanismos de modularidad, que permiten independizar unas partes del programa de otras.
  3. Mecanismos de seguridad estática: los sistemas de tipos protegen de errores triviales que de otro modo pasarían inadvertidos. A la vez tienen la suficiente flexibilidad para no constreñir excesivamente al programador.
  4. Potenciamiento de la reutilización: los tipos polimórficos, la sobrecarga, la genericidad, la modularidad y el orden superior, potencian la creación de componentes reutilizables, bien tal cual son, bien previa adaptación.
- En mi opinión, estas tendencias se van a mantener en el futuro porque han demostrado ser capaces de mantener bajo control los siempre crecientes complejidad y volumen de los sistemas software.
- Pero no se debe bajar la guardia y volver a escribir los programas de modo artesanal, como está ocurriendo con las tecnologías asociadas a Internet.

El futuro (2)

Unas notas finales sobre el papel que estimo jugarán los futuros compiladores:

- Hasta ahora estos se han ocupado fundamentalmente de garantizar la corrección sintáctica y de tipos de los programas, además de haber puesto un énfasis comprensible en generar un código lo más eficiente posible.
- Pero les aguardan tareas de mayor responsabilidad, en base a los avances de los últimos años en las técnicas de análisis estático:
  - demostrar la terminación de bucles
  - sintetizar invariantes para los mismos
  - estimar cotas superiores al coste en tiempo y en memoria de los programas
  - demostrar la ausencia de bloqueo de los programas concurrentes

El futuro (2)

Unas notas finales sobre el papel que estimo jugarán los futuros compiladores:

- Hasta ahora estos se han ocupado fundamentalmente de garantizar la corrección sintáctica y de tipos de los programas, además de haber puesto un énfasis comprensible en generar un código lo más eficiente posible.
- Pero les aguardan tareas de mayor responsabilidad, en base a los avances de los últimos años en las técnicas de análisis estático:
  - demostrar la terminación de bucles
  - sintetizar invariantes para los mismos
  - estimar cotas superiores al coste en tiempo y en memoria de los programas
  - demostrar la ausencia de bloqueo de los programas concurrentes

CODA

- Los programas son entidades tremendamente complejas, pero a diferencia de otros productos de la ingeniería, son absolutamente predecibles y no sufren desgaste por el uso.
- Nuestra obligación es dominarlos e impedir que ellos nos dominen.

El futuro (2)

Unas notas finales sobre el papel que estimo jugarán los futuros compiladores:

- Hasta ahora estos se han ocupado fundamentalmente de garantizar la corrección sintáctica y de tipos de los programas, además de haber puesto un énfasis comprensible en generar un código lo más eficiente posible.
- Pero les aguardan tareas de mayor responsabilidad, en base a los avances de los últimos años en las técnicas de análisis estático:
  - demostrar la terminación de bucles
  - sintetizar invariantes para los mismos
  - estimar cotas superiores al coste en tiempo y en memoria de los programas
  - demostrar la ausencia de bloqueo de los programas concurrentes

CODA

- Los programas son entidades tremendamente complejas, pero a diferencia de otros productos de la ingeniería, son absolutamente predecibles y no sufren desgaste por el uso.
- Nuestra obligación es dominarlos e impedir que ellos nos dominen.

¡MUCHAS GRACIAS!