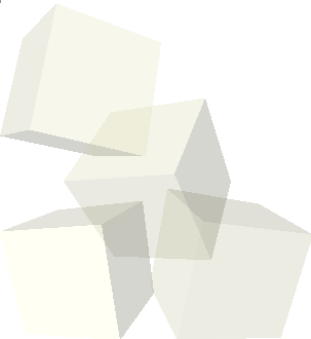




Nuevas Perspectivas en Lenguajes de Programación:

Teoría de Tipos y Seguridad de Tipos

Borja Sotomayor
10 de julio de 2006



1



Charla organizada por:



Cursos de Julio 2006

<http://www.eside.deusto.es/eventos/cursillo/>



2





Disclaimer

- No soy un experto en Teoría de Lenguajes de Programación (PL). Esta charla es el resultado de (1) haber estudiado PL en Chicago y (2) incontables conversaciones y charlas con verdaderos expertos en PL.
- Agradecimientos: Jacob Matthews, Mike Rainey, y Adam Shaw
- Esta no es una charla dogmática.
- Más detalles sobre mi (verdadera) área de investigación (Grid Computing) en mi web:
 - <http://people.cs.uchicago.edu/~borja/>

3



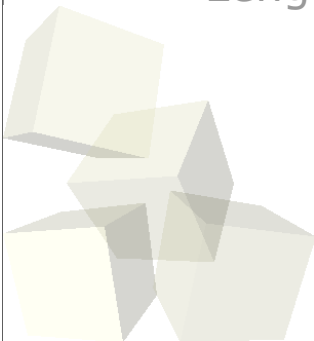
Índice

- Introducción
- Comprobaciones estáticas vs. dinámicas
- Teoría de Tipos y Seguridad de Tipos
- El estado del arte
- Lenguajes Interesantes

4



- Introducción
- Comprobaciones estáticas vs. dinámicas
- Teoría de Tipos y Seguridad de Tipos
- El estado del arte
- Lenguajes Interesantes

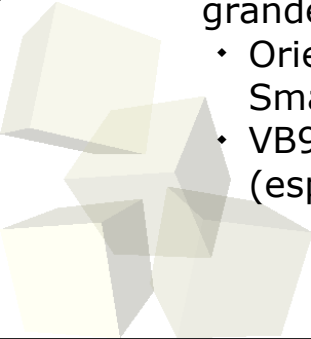


- Los lenguajes de programación han evolucionado durante décadas.
- Hay un puñado de lenguajes que se han convertido en los “más populares” para ingeniería:
 - C/C++, Java, C#, Visual B***c, Delphi, ...
 - Python, Perl, PHP, Java/ECMAScript, ...
- En círculos académicos:
 - Haskell, Scheme, ...





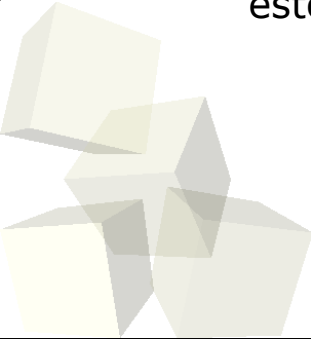
Introducción (II)

- Sin embargo, hay *miles* de lenguajes de programación [1].
 - Muchos de estos lenguajes son *research languages* que, en muchas ocasiones, no llegarán a ser adoptados por la industria informática.
 - Sin embargo, exploran técnicas y nuevos paradigmas que eventualmente llegan a los grandes lenguajes de programación.
 - Orientación a objetos: surgió de Simula y Smalltalk
 - VB9 influido por programación funcional (especialmente Haskell) [2]
- 

7



Introducción (III)

- Es interesante conocer cuales son estos *research languages* porque:
 - Las ideas incluidas en esos lenguajes pueden acabar llegando a lenguajes más populares.
 - En algunos casos concretos, puede resultar más eficiente utilizar uno de estos lenguajes.
- 

8

Introducción (IV)

- ¿Cuales son las principales preocupaciones de estos lenguajes?
- Hay muchas subareas, pero podemos distinguir dos importantes áreas generales:
 - La detección estática de bugs
 - Lenguajes para afrontar nuevos desafíos

9

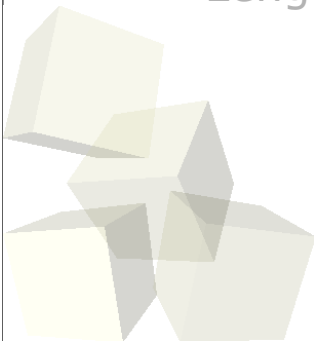
Introducción (V)

- Objetivos de esta charla
 - Presentar nuevas perspectivas en lenguajes de programación, para ver que hay un universo de lenguajes más allá del C/C++, Java, etc. “de toda la vida”.
 - Nos centraremos en la diferencia entre comprobaciones estáticas (*static checking*) y comprobaciones dinámicas (*dynamic checking*).
 - Énfasis en área concreta llamada Teoría de Tipos (*Type Theory*) y la Seguridad de Tipos (*Type Safety*).
 - Comentar brevemente algunos lenguajes que aplican las ideas expuestas en la charla.

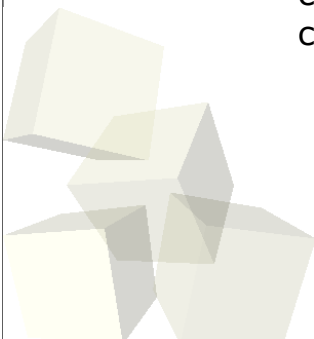
10



- Introducción
- Comprobaciones estáticas vs. dinámicas
- Teoría de Tipos y Seguridad de Tipos
- El estado del arte
- Lenguajes Interesantes



- Comprobación estática
 - Realizada por el compilador al analizar el código fuente.
 - Un compilador no sólo detecta errores de sintaxis, también puede detectar ciertos errores de lógica.
 - P.ej. El compilador de Java puede rechazar código que es *sintácticamente* válido pero que contiene errores básicos de lógica.



Comprobación Estática (II)

```
public int sumar(int a, int b)
{
    int c = a + b;
}
```

No hay return



```
public int sumar(int a, int b)
{
    int c;
    return a+b;
}
```



Variable
no utilizada

```
public void refreshGUI()
{
    Vector newButtons;

    for(int i=0; i<newButtons.size(); i++)
    {
        // ...
    }
}
```

Objeto no inicializado



13

Comprobación Dinámica (I)

- Comprobación dinámica
 - Realizada en tiempo de ejecución (responsabilidad del programador)
 - Hay comprobaciones que el compilador no puede realizar.
 - P.ej. errores de lógica específicos a nuestro dominio. P.ej. "Comprobar que los strings que representan contraseñas nunca tienen más de ocho caracteres"
 - Sin embargo, hay muchas comprobaciones dinámicas que pueden pasar a ser responsabilidad del compilador. P.ej. Colecciones en Java 1.4 vs. Java 5

14

Comprobación Dinámica (II)

```
public void despedir(Vector vectorEmpleados)
{
    for(int i=0; i<vectorEmpleados.size(); i++)
    {
        Empleado empl = (Empleado) vectorEmpleados.get(i);
        empl.despedir();
    }
}
```

¿Y si antes de llamar a despedir()
he hecho esto?



```
vectorEmpleados.add(new Integer(5))
```

¡ClassCastException!

15

Comprobación Dinámica (III)

- Solución:
 - El programador es el responsable de añadir comprobaciones para asegurarse de que esos errores son manejados adecuadamente.
 - No es una solución ideal.
 - El programador puede olvidarse de añadir esas comprobaciones (io puede darle pereza!)
 - El código suele ser engorroso y dificulta la lectura del resto del código.

16

Comprobación Dinámica (IV)

```
public void despedir(Vector vectorEmpleados)
{
    for(int i=0; i<vectorEmpleados.size(); i++)
    {
        Object obj = vectorEmpleados.get(i);
        if(obj instanceof Empleado)
        {
            Empleado empl = (Empleado) obj;
            empl.despedir();
        }
        else
        {
            System.err.println("Vector de empleados
                contiene objeto no-empleado");
        }
    }
}
```

17

Comprobación Dinámica (V)

- Además, hay ciertos errores para los que no es posible añadir comprobaciones. Son bugs con los que tendremos que pelearnos al depurar nuestro programa.

```
public void despedir(Vector vectorEmpleados)
{
    for(int i=0; i<=vectorEmpleados.size(); i++)
    {
        ...
    }
}
```

iArrayIndexOutOfBoundsException!



18

Java 1.4 vs. Java 5

- En Java 5 esta responsabilidad pasa del programador (comprobación dinámica) al compilador (comprobación estática).
 - ClassCastException -> Templates
 - ArrayIndexOutOfBoundsException -> Bucle 'for' mejorado

```
public void despedir(Vector<Empleado> vectorEmpleados)
{
    for (Empleado e: vectorEmpleados)
        e.despedir();
}
```

19

Otros errores (I)

- Pero todavía hay muchos errores que el compilador no detectará.
 - ¿Y si vectorEmpleados es null?
 - ¿Y si uno de los valores del vector es null?
- Solución: ¿Capturar NullPointerException en todos lados? ¡Locura!
- Sin embargo, hay lenguajes que están diseñados para capturar este tipo de errores estáticamente.
 - Tipos "option" en SML, OCaml, ...
 - Tipos "maybe" en Haskell

20

Otros errores (II)

- En Java, todos los tipos son *nullable* (excepto los tipos primitivos)
 - Es imposible indicar que una variable del tipo `FOOBAR` *nunca* debe tener un valor nulo.
- En SML, esto es posible gracias a los tipos *option*

```
type pair = int * int      Pareja de enteros
type pair = (int * int) option  (o NULL)
```

21

Otros errores (III)

```
type empleado = { nombre : string, dni : int, empl : bool };
type empleadoList = empleado list;
```

```
fun despedirEmpleado ( e:empleado ) =
  { nombre = #nombre e, dni = #dni e, empl = false }
```

```
fun despedir emps =
  map despedirEmpleado emps
```

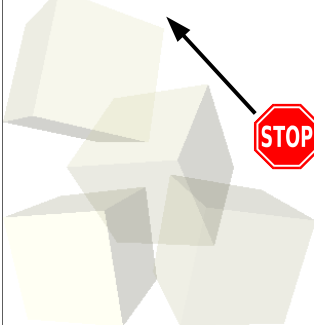
22

Otros errores (IV)

```
type empleado = { nombre : string, dni : int, empl : bool };
type empleadoList = empleado option list;

fun despedirEmpleado ( e:empleado ) =
  { nombre = #nombre e, dni = #dni e, empl = false }

fun despedir emps =
  map despedirEmpleado emps
```



Si en el programa llamamos a “despedir” sobre una lista de tipo “empleadoList”, el compilador puede inferir que se puede producir un “null pointer exception” en despedirEmpleado, ya que no contemplamos el caso NULL.

23

Otros errores (V)

```
type empleado = { nombre : string, dni : int, empl : bool };
type empleadoList = empleado option list;

fun despedirEmpleado ( SOME (e:empleado) ) =
  SOME { nombre = #nombre e, dni = #dni e, empl = false }
| despedirEmpleado (NONE) =
  (print "WARN: Empleado nulo"; NONE)

fun despedir emps =
  map despedirEmpleado emps
```

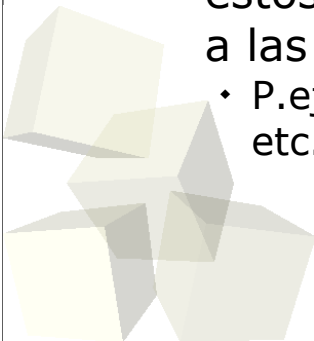


24



Resumen

- Recordando: se están desarrollando lenguajes que la detección de bugs pasa de ser una comprobación dinámica a ser una comprobación estática.
- Las técnicas y ideas exploradas en estos lenguajes eventualmente llegan a las masas.
 - P.ej. Orientación a objetos, Templates, etc.

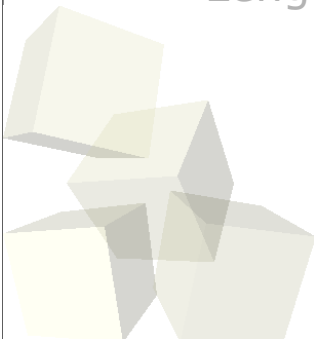


25



Índice

- Introducción
- Comprobaciones estáticas vs. dinámicas
- Teoría de Tipos y Seguridad de Tipos
- El estado del arte
- Lenguajes Interesantes



26

Teoría de Tipos (I)

- La Teoría de Tipos [3] es una rama de PL que se centra en el estudio de los *type systems* (“el sistema de tipos”) en los lenguajes de programación.
 - Los tipos no son una parte trivial (“int”, “float”, etc.) de un lenguaje de programación.
- *Tipo*: Clasificación de un valor o una expresión según el tipo que computa.
- Todo lenguaje tiene un sistema de tipos (aunque sea para especificar que hay un único tipo variable)

27

Teoría de Tipos (II)

- Un sistema de tipos bien diseñado permite realizar más comprobaciones estáticas.
- ¿Qué es un sistema de tipos bien diseñado?
 - ¿“Fuertemente tipado”?
 - ¿Tipado estático o dinámico?
 - ¿Añadir el máximo posible de comprobaciones en el compilador para evitar bugs frecuentes?

28

Seguridad de Tipos (I)

- En general, un sistema de tipos debe aspirar a ser *type-safe* (debe garantizar la "seguridad de tipos") [3,4]
- En un lenguaje *type-safe* el compilador no aceptará *ningún* programa mal tipado.
 - Expresiones del tipo "Hola" / 5
 - Expresiones con errores más sutiles como los que hemos visto antes.
 - El compilador también puede rechazar programas que, aun estando mal tipados, no generarían una excepción en ejecución.

```
if (true)
    return "Hola";
else
    return 42;
```

29

Seguridad de Tipos (II)

- La seguridad de tipos se puede definir formalmente, y se puede demostrar que un lenguaje es *type-safe*.
 - *Progreso*: Un término bien tipado (una "expresión") nunca se queda atascado. Es decir, o es un valor, o podemos tomar un paso de evaluación.
 - *Preservación*: Si tomamos un paso de evaluación sobre un término bien tipado, el término sigue estando bien tipado.
- Esta demostración se realiza inductivamente sobre todos los posibles términos de nuestro lenguaje.

30

Seguridad de Tipos (III)

- Ventaja de estos lenguajes: tenemos una garantía *formal* de que, una vez compilado (realizada la comprobación estática), hay ciertas operaciones que nuestro programa no realizará (operaciones que podrían resultar en un fallo)
- Sólo hay un lenguaje "complejo" para el que exista una demostración de que es type-safe: SML
- Se sospecha que Java 5 es type-safe, pero realizar la demostración sería una tarea hercúlea

31

Ventajas de SML (I)

- Entonces... ¿qué tiene SML que no tenga Java?
 - Un sistema de tipos mucho más compacto y mejor diseñado con el que se pueden realizar demostraciones formales.
 - Java deja demasiados errores en manos de excepciones que son lanzadas por el sistema, no por el usuario.
 - El sistema de tipos tiene algunas características bastante útiles.

32

Ventajas de SML (II)

■ Tipos “producto” (“tuplas”)

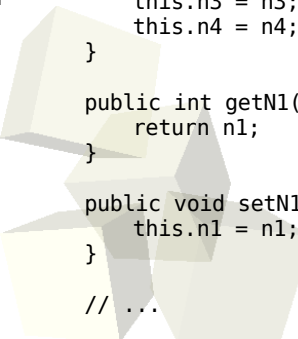
- Cuando necesitamos tuplas en nuestro programa, no tenemos que declarar una clase específicamente para esa tupla.

```
public class tupla4 {  
    private int n1,n2,n3,n4;  
  
    public tupla4(int n1, int n2, int n3, int n4)  
    {  
        this.n1 = n1;  
        this.n2 = n2;  
        this.n3 = n3;  
        this.n4 = n4;  
    }  
  
    public int getN1() {  
        return n1;  
    }  
  
    public void setN1(int n1) {  
        this.n1 = n1;  
    }  
  
    // ...  
}
```

← Java

SML

type 4tupla = int * int * int * int



33


Ventajas de SML (III)

■ Inferencia de tipos

- SML no necesita anotaciones de tipos. Es capaz de deducir el tipo de cada expresión analizando el programa entero.


```
fun incr a = a + 1
```

int -> int



```
fun incr a = a + 1.0
```

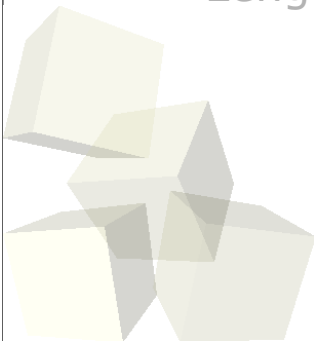
real -> real



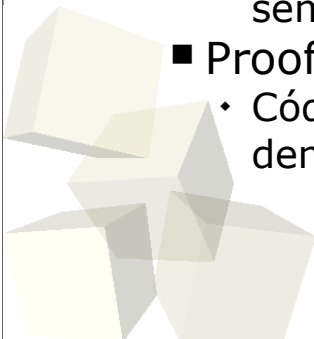
34



- Introducción
- Comprobaciones estáticas vs. dinámicas
- Teoría de Tipos y Seguridad de Tipos
- El Estado del Arte
- Lenguajes Interesantes

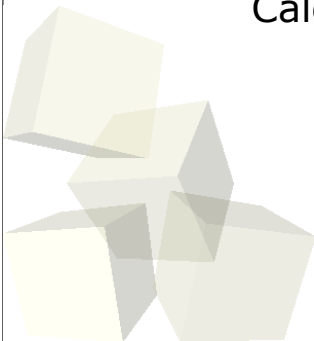


- Software Transactional Memory [5]
 - Transacciones a nivel de lenguajes de programación (orientado a memoria compartida)
- Semantics Preservation
 - Garantizar que las optimizaciones que introduce un compilador no alteran la semántica original del programa.
- Proof-Carrying Code
 - Código que lleva adjunto una demostración de que el código es seguro.





- **Typed Assembly**
 - Código ensamblador con tipos
- **Sistemas concurrentes**
 - Integrar soporte para concurrencia directamente en el lenguaje (no en la API).
 - Modelos formales: Pi Calculus, Ambient Calculus.

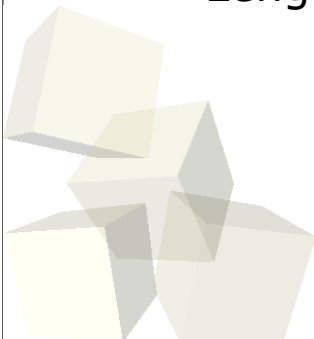


37



Índice

- Introducción
- Comprobaciones estáticas vs. dinámicas
- Teoría de Tipos y Seguridad de Tipos
- El Estado del Arte
- Lenguajes Interesantes



38



Lenguajes Interesantes (I)

- SML
 - <http://www.smlnj.org/> (SML/NJ)
 - Type-safe
 - Especialmente indicado para escribir compiladores.
 - Principal carencia: librerías
- OCaml
 - <http://caml.inria.fr/>
 - Desciende, al igual que SML, del lenguaje ML [8]
 - Nadie ha demostrado si es type-safe, pero tiene un sistema de tipos muy similar al de SML.
 - Dispone de muchas librerías.
- Cyclone
 - <http://cyclone.thelanguage.org/>
 - C con seguridad de tipos
 - Toda la versatilidad y potencia de C, pero con muchas más comprobaciones estáticas que un compilador de C.

39



Lenguajes Interesantes (II)

- XDuce
 - <http://xduce.sourceforge.net/>
 - Lenguaje type-safe para procesamiento de XML
- Scheme
 - Basado en LISP, aunque permite tanto el paradigma funcional como el procedural.
 - Tipado dinámico.
- J Wig
 - <http://www.brics.dk/JWIG/>
 - Buen ejemplo de un lenguaje de programación diseñado para afrontar un nuevo reto (programación web)

40

¿Preguntas?

Borja Sotomayor
Department of Computer Science
University of Chicago
borja@cs.uchicago.edu
<http://people.cs.uchicago.edu/~borja/>

41

Bibliografía

- [1] "List of Programming Languages" http://en.wikipedia.org/wiki/List_of_programming_languages
- [2] "Confessions of a Used Programming Language Salesman: Getting the Masses Hooked on Haskell". Eric Meijer. Submitted to ICFP2006. <http://research.microsoft.com/~emeijer/Papers/ICFP06.pdf>
- [3] "Types and Programming Languages". Benjamin C. Pierce. 2002, The MIT Press.
- [4] "Type Safety". <http://en.wikipedia.org/wiki/Type-safety>
- [5] "Software Transactional Memory". http://en.wikipedia.org/wiki/Software_transactional_memory
- [6] Proof-Carrying Code. http://en.wikipedia.org/wiki/Proof-Carrying_Code
- [7] Typed Assembly Language (TAL). <http://www.cs.cornell.edu/talc/>
- [8] ML Programming Language. http://en.wikipedia.org/wiki/ML_programming_language
- [4] "The Next Mainstream Programming Language: A Game Developer's Perspective." Tim Sweeney. POPL 2006. <http://www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt>

42