PRÁCTICA 1
MAC

El problema de la parada

Juan Antonio Velasco Gómez
Miguel Sánchez Maldonado

1. Muestre que, si pudiésemos resolver el problema de la parada, también podríamos resolver muchos problemas matemáticos no resueltos. Ilustre cómo se podría demostrar la conjetura de Goldbach si tuviésemos un algoritmo que resuelva el problema de la parada.

> *Conjetura de Goldbach: Todo número par mayor que 2 puede descomponerse como la suma de dos números primos.*

Solución

Proponemos una solución que vaya comprobando números tomando como cierto el problema de la parada y siempre podremos comprobar todos los números.

Gödel vió que hay cosas que son verdaderas y no se pueden demostrar. Vamos a buscar una contradicción suponiendo que todas las cosas verdaderas se pueden demostrar. Por tanto tendríamos un programa que sería capaz de demostrar todo lo verdadero.

Convertimos el programa y la entrada de datos en números enteros y buscamos un algoritmo que para el programa y la entrada anterior, nos diga si el programa para o no. Esto resolvería el problema de la parada puesto que nos diría si el programa para o no. De forma que llegamos a una contradicción puesto que no podemos resolver el problema de la parada.

2. A partir de los resultados de Turing sobre el problema de la parada, demuestre el Teorema de Incompletitud de Gödel: Dado cualquier conjunto consistente y computable de axiomas, existe una sentencia verdadera sobre números enteros que no puede demostrarse a partir de dichos axiomas. Consistente hace referencia a que no se puede derivar ninguna contradicción, mientras que computable quiere decir que existe un número finito de axiomas o bien, aunque haya un número infinito, existe un algoritmo que nos permite generarlos (si no incluyésemos este requisito, podríamos partir de un conjunto de axiomas formado por todas las sentencias verdaderas sobre números enteros; lo cual, de todas formas no ser´ıa demasiado útil en la práctica).

Solución:

Supongamos que es falso, es decir, que toda sentencia verdadera sobre números enteros puede demostrarse. Es decir un programa que nos devuelva verdadero o falso.

Por este hecho podemos computar el problema, de modo que tenemos un “demostrador” de sentencias, que nos dice si toda sentencia es verdadera o falsa.

Ahora, codificamos el problema de la parada con la hipótesis de que para con un número entero y se lo introducimos a nuestro programa. Pero esa sería la contradicción porque el problema de la parda es no decidible.

3. A partir de los resultados de Turing sobre el problema de la parada, demuestre el Segundo Teorema de Incompletitud de Gódel: Si una teoría F es consistente, entonces F no puede demostrar su propia consistencia. En otras palabras, las únicas teorías matemáticas que pueden demostrar su consistencia son precisamente aquéllas que no son consistentes.

Solución

Para este demostración partimos de los resultados de Turing, nos vamos a basar en los resultados del ejercicio 2, es decir a partir de los resultados de Turing para el problema de la parada se puede demostrar el primer Teorema Incompletitud de Gödel. Con este primer teorema nos basamos para demostrar el segundo Teorema de Incompletitud de Gödel.
Por tanto como conclusión tenemos que si partimos de los resultados de Turing vamos a poder demostrar el segundo teorema de Incompletitud de Gödel, aunque para la demostración nos tenemos que apoyar en el primer teorema de Incompletitud de Gödel y por tanto por transitividad,
Turing -> 1º Teorema Incompletitud
1º Teorema Incompletitud -> 2º Teorema Incompletitud
Turing -> 2º Teorema Incompletitud

La demostración del segundo teorema la podemos sacar del siguiente enlace
http://enciclopedia.us.es/index.php/Teorema_de_la_incompletitud_de_G%C3%B6del
 Esbozo de prueba del segundo teorema
Sea P la sentencia indecidible construida previamente, y asumamos que la consistencia del sistema se puede probar dentro del propio sistema. Hemos visto arriba que si el sistema es consistente, entonces P no es demostrable. La prueba de esta implicación se puede formalizar en el propio sistema, y por tanto la afirmación “P no es demostrable”, o no “P(p)” se puede demostrar en el sistema
Pero esta última declaración es equivalente a P mismo(y esta equivalencia se puede demostrar en el sistema), de modo que p se puede demostrar en el sistema. Esta contradicción pone de manifiesto que el sistema debe ser inconsistente.

4. Demuestra, por reducción al problema de la parada, que no es posible desarrollar un depurador automático que sea capaz de, en general:

- **Determinar si un programa incluye algún bucle infinito.**
- **Determinar si un programa mostrará por pantalla un mensaje determinado.**
- **Determinar si un programa llegará a ejecutar una sentencia determinada.**
- **Determinar si una variable se inicializará siempre antes de ser utilizada.**

Solución

Vamos a ir viendo uno a uno todos los apartados del ejercicio.

En el primer apartado, si tiene un bucle infinito, el depurador automático que lo determina entraría en un bucle infinito y no pararía. Si fuera capaz de indicarlo, el programa pararía. Pero sabemos por el problema de la parada que esto no es posible.

En el segundo apartado, partimos de un programa, si el mensaje que queremos mostrar es el nuestro, entonces para. Donde pueda parar le metemos un bucle infinito. El programa para si muestra el mensaje y no para si no lo muestra que se corresponde con el problema de la parada. Para completar la demostración hay que comprobarlo para cualquier mensaje.

Si el mensaje que queremos mostrar es el propio mensaje del programa, si lo encuentra para. Escribir un programa que va enumerando todos los programas posibles.

Tomamos un programa que muestra el texto, si lo muestra, no hacemos nada. Si no lo muestra, lo imprimimos nosotros.

Creamos un programa que nos dice si un mensaje se muestra a sí mismo o no:

```
mensaje(p,m)
for all p
        if !muestra (p,p)
                print p
```

Pero este bucle generaría una contradicción.

En el tercer apartado creamos el programa Ejecuta(p,m) que nos dice si p ejecuta la sentencia m

```
F(p)
if !Ejecuta(p,p)
        exe(p);
```

```
F(F)                    //Le pasamos el propio programa
```

este programa genera una contradicción en sí misma porque cuando no ejecuta, lo ejecuta.

En el cuarto apartado,tomamos un programa que nos dice si una variable a se inicializa.

```
Q(p)
if !Inicializa(p,a)
        new a=12;
a = a+1;
Q(Q);
```

Llegamos entonces a una contradicción porque cuando no inicializa, lo inicializa.

Esto no quiere decir que no se puedan construir sistemas en los que sea posible realizar demostraciones sobre el comportamiento de un programa, sólo que no existen soluciones universales a tales problemas. Por ejemplo, la máquina virtual de Java está diseñada para poder permitir determinados tipos de demostraciones y un compilador de Java es capaz de detectar muchos de estos errores en tiempo de compilación. Busque información en Internet sobre el tema.

Solución

La máquina virtual de Java es el entorno en el que se ejecutan los programas Java, la misión principal es garantizar la portabilidad de las aplicaciones. Define esencialmente un ordenador abstracto y especifica las instrucciones (bytecodes) que este ordenador puede ejecutar.

En Java existen los errores de compilación y las advertencias (warnings), a través de Eclipse podemos detectar errores y advertencias al mismo tiempo que escribimos. Además para podemos indicar a Eclipse que incremente el nivel de advertencias/errores en gran diversidad de casos.Esto se hace en el menú de Preferences / Java / Compiler / Errors.

Por otra parte tenemos herramientas como PMD, las cuales analizan el código en busca de errores de más alto nivel que los que detecta el compilador, nos detectan posibles bugs debidos a try/catch o switch vacíos, códigos que no se alcanzan o variables y parámetros que no se usan. Los errores de compilación se pueden clasificar:

Errores de sintaxis: el código tecleado no cumple las reglas sintácticas del lenguaje Java, por ejemplo, falta un punto y coma al final de una sentencia o se teclea mal el nombre de una variable (que había sido declarada con otro nombre).

Errores semánticos: código que, siendo sintácticamente correcto, no cumple reglas de más alto nivel, por ejemplo imprimir el valor de una variable a la que no se ha asignado valor tras declararla:

```
1   public void funcion()
2   {
3       int a;
4       Console.println(a);
5   }
6
```

Errores en cascada: no son otro tipo de error, pero son errores que confunden al compilador y el mensaje que éste devuelve puede indicar la causa del error lejos de donde realmente está.

Otro problema que crea confusión con respecto a la localización del error son las llaves mal cerradas. Esto se debe a que el compilador de Java no tiene en cuenta la indentación de nuestro código. Mientras que el programador puede ver, a través de la indentación, dónde falta cerrar la llave de una función, bucle o clase, el compilador podría darse cuenta al terminar de leer el archivo e indicarlo ahí.

5. Muestre que no es posible determinar, en general, si dos programas hacen lo mismo. Esto es, dados dos programas P y Q, demuestra que no es decidible el problema de determinar si ambos calculan la misma función: P(x) = Q(x) para todo x.

Observe que esto implica que, en general, no se puede determinar si un programa original P y su versión optimizada P' producida por un compilador calculan exactamente lo mismo.

<u>Solución</u>

No podemos decidir si ambos programas calculan la misma función.Si esto fuese posible estaríamos resolviendo el problema de la parada, y por tanto el problema de la parada es decidible y sería contradicción.
Como solución lo podemos ver en un algoritmo en pseudocodigo

```
F(P,Q)
if( P(x) = Q(x))
        Q(x)=Q(x)+5
else
        P(x)=Q(x)
```

F(F,F)
Si son iguales los hago distintos y si son distintos los hago iguales y los vuelvo a llamar.

6. Usando el resultado del ejercicio anterior, muestre cómo siempre se puede derrotar a un troyano o caballo de Troya que intente pasar desapercibido (un troyano es un software malicioso que se presenta al usuario como un programa aparentemente legítimo e inofensivo, pero que, al ejecutarlo, le brinda a un atacante acceso remoto al equipo infectado mediante la creación de una puerta trasera o backdoor).

Ken Thompson ('Reflections on Trusting Trust', 1984, http://cm.bell-labs.com/who/ken/trust.html) defendía que podía colocar un troyano en un compilador de C que compilar ıa mal el login para permitirle acceder a cualquier sistema Unix que se compilase con su compilador de C y que, además, podía compilar mal el compilador de C para insertar una copia del troyano sin que el troyano en sí apareciese en el código fuente del compilador de C. Demuestre que estaba equivocado.

Solución

Para esta demostración vamos a suponer que es cierto para llegar a una contracción. Suponemos que vamos a compilar el programa P en el compilador donde se encuentra el troyano, y por otro lado P' el cual es compilado en el compilador que no tiene troyano Vamos buscando que ocurra P(x) = P'(x) pero si esto ocurre es porque los dos compiladores son iguales, es decir el que tiene troyano y el que no, y por tanto no podríamos detectar el troyano o en este caso el compilador afectado por el troyano.

Por el ejercicio anterior sabemos que será indecidible y por tanto sería una contradicción de lo que hemos supuesto. En nuestro caso que el troyano si sería detectado.

7. Demuestre que no existe ningún algoritmo que sea capaz de determinar si un problema matemático es decidible o no (esto es, un programa que sea capaz de indicarnos si una sentencia es decidible, sin llegar determinar si es cierta o falsa).

Solución

Para ver si existe un algoritmo capaz de determinar si un problema matemático es decidible o no, partimos de una máquina que posee todos las demostraciones del mundo y que es capaz de determinar con ellas si un problema es decidible o no.

Si le pasamos como parámetro la sentencia "El programa para", no puede demostrarlo, puesto que sabemos que el problema de la parada es no decidible, pero no sabemos con ello nada de un caso particular.

8. Muestre cómo se podría resolver el problema de la parada cuando imponemos una restricción sobre la cantidad de memoria que puede utilizar un programa. Obviamente, necesitará un sistema que disponga de muchísima más memoria de la que permitimos que utilice el programa...

Solución

Si ponemos una restricción sobre la cantidad de memoria que puede utilizar un programa para resolver el problema de la parada bastaría con saber esa cantidad de memoria.

Si cuando se llegue a esa cantidad de memoria el programa no ha parado, suponemos que es indecidible mientras que si ha parado antes de llegar a ese tope de memoria, entonces podemos concluir que el problema es decidible.