Computabilidad

Horst H. von Brand
vonbrand@inf.utfsm.cl

Departamento de Informática
Universidad Técnica Federico Santa María

Contenido

Límites de la computación

Hemos analizado varios modelos restringidos de «computación», explorando sus capacidades y limitaciones.
Ahora queremos ver el problema en forma global: ¿qué puede (o no) hacer un algoritmo? Nos interesa qué se puede hacer *en principio*, omitiendo límites de uso de memoria y tiempo de ejecución. Más adelante analizaremos situaciones algo más realistas.

Límites de la computación

Por ahora usaremos como modelo de computación programas en C idealizado: tipos básicos son únicamente enteros (de verdad, sin limitaciones de valor), punteros y caracteres. No consideramos posibles límites de memoria o tamaño del programa. Nos interesan programas que efectúan manipulaciones finitas de objetos discretos. Podemos representar aproximaciones de números reales como fracciones, de requerirlos. De todas formas, no podemos operar con números reales en tiempo finito más que en casos muy especiales.

Un programa clásico

```c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}
```

¿Escribe «¡Hola, mundo!» este programa?

Parece una pregunta trivial.

Resulta que no lo es.

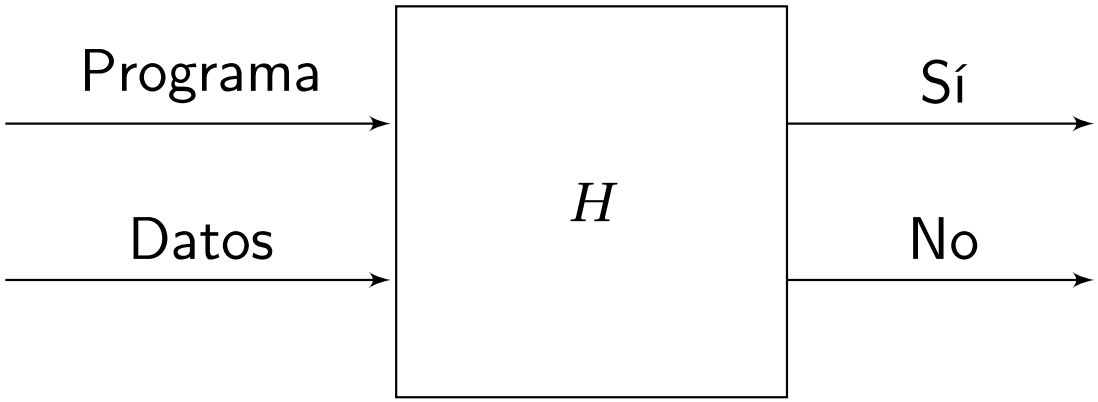
¿Escribe «¡Hola, mundo!» este programa?

Dado un programa C como descrito, en un formato acordado de antemano (por ejemplo, un *tarball* de los fuentes que incluye una `Makefile` para compilarlo), llamémosle P; junto a un archivo de entrada al programa P, llamémosle D.
En esos términos, nuestro problema es: «Si se ejecuta el programa P con datos D, ¿escribe "`¡Hola, mundo!`" como primera salida?»

Vemos que para nuestro programa clásico la respuesta es «Sí»: No lee su entrada, y escribe «`¡Hola, mundo!`». Habrán otros programas para los cuales la respuesta «No» es igual de trivial. Pero la pregunta es respecto de un programa cualquiera, posiblemente mucho más complicado.

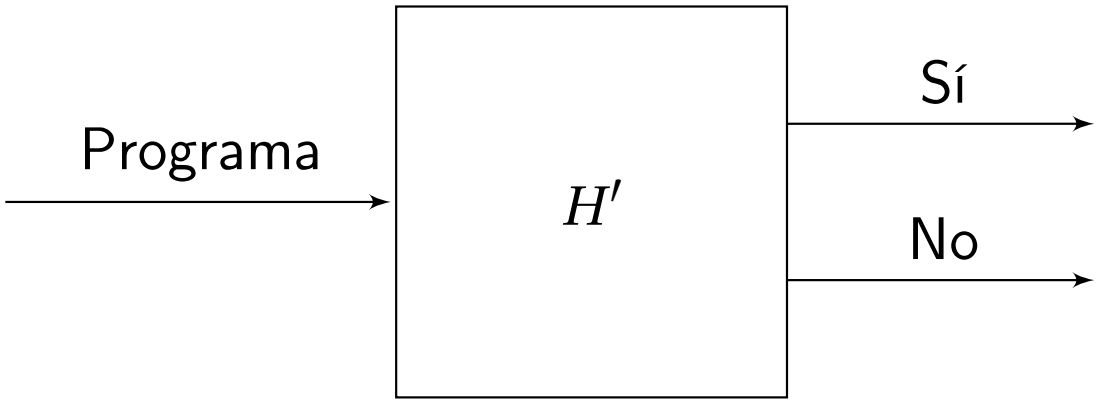
Imposibilidad de resolver «¡Hola, mundo!»

La demostración es por contradicción. Supongamos que hay un programa hipotético H que resuelve el problema «¡Hola, mundo!». El programa viene dado como fuentes C, en algún formato acordado (como ser un tarball), y a su vez procesa programas en esa misma presentación. El siguiente diagrama esquematiza H.



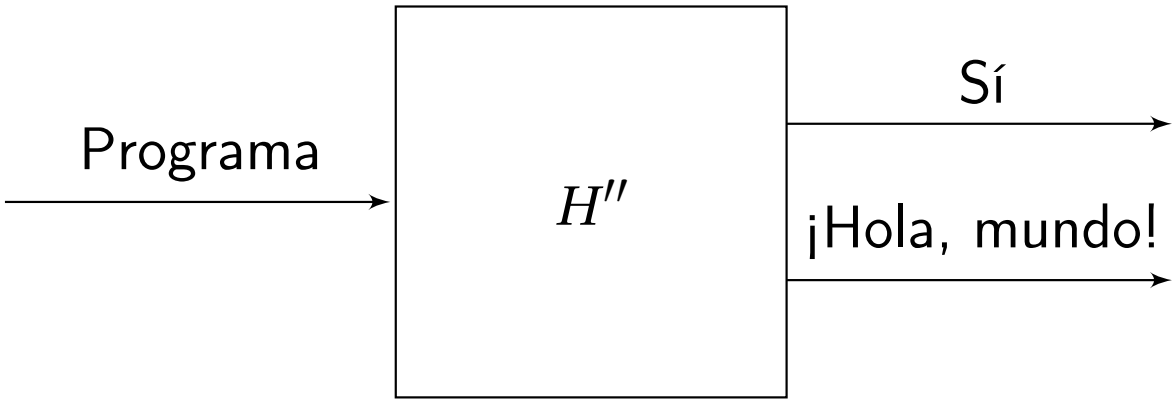
Imposibilidad de resolver «¡Hola, mundo!»

Como primer paso, modificaremos H para solo tomar un programa como entrada y considerar el resultado de ejecutarlo sobre sus fuentes. Esto se logra en C simplemente duplicando el archivo de programa como archivo de datos. Llamaremos H' al resultado.



Imposibilidad de resolver «¡Hola, mundo!»

Como segundo paso modificamos H' para que en vez de responder «No» escriba «¡Hola, mundo!», le llamaremos H''. La modificación es almacenar lo que se desea escribir y revisar si es «No», en tal caso escribir «¡Hola, mundo»; si es «Sí» escribir esto.



Imposibilidad de resolver «¡Hola, mundo!»

Si alimentamos H'' con los fuentes de H'' vemos lo siguiente:

- Si H'' con datos H'' escribe «Sí», debiera escribir «¡Hola, mundo!».
- Si H'' con datos H'' escribe «¡Hola, mundo!», debiera escribir «Sí».

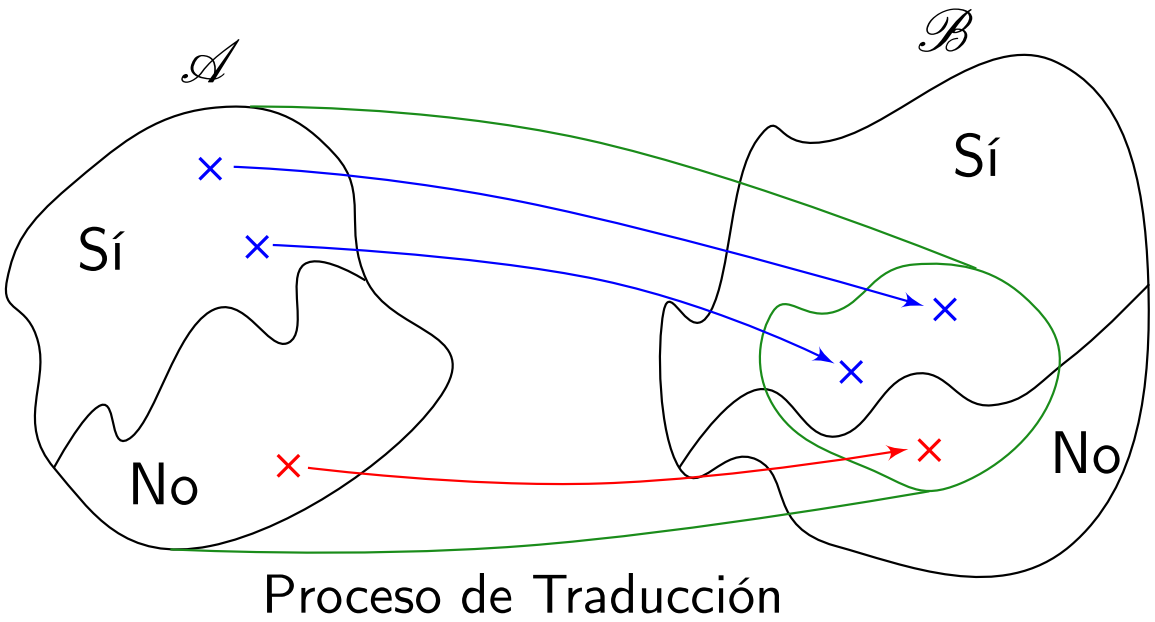
El comportamiento de H'' es contradictorio, tal programa no puede existir. Como construimos H'' a partir del hipotético H, tampoco puede existir H.

Reducción de problemas

Este importante concepto es lo que hacemos rutinariamente al resolver problemas en todos los ámbitos: enfrentados a un problema, lo traducimos en un problema que sabemos resolver y trabajamos éste.

Sean dados problemas $\mathscr{A}$ y $\mathscr{B}$. Si toda instancia a del problema $\mathscr{A}$ se traduce en la instancia b del problema $\mathscr{B}$ tal que b tiene la misma respuesta que a, diremos que $\mathscr{A}$ *se reduce a* $\mathscr{B}$, anotaremos $\mathscr{A} \leq \mathscr{B}$. Note que $\leq$ entre problemas es un preorden (relación transitiva y reflexiva). Esto representa que el problema $\mathscr{B}$ no es más fácil que $\mathscr{A}$: si tenemos cómo resolver todas las instancias de $\mathscr{B}$, $\mathscr{A}$ es pan comido. Pero puede ser que las instancias de $\mathscr{B}$ en las que se traduce $\mathscr{A}$ sean solo casos muy especiales, saber cómo resolver $\mathscr{A}$ no nos ayuda a resolver $\mathscr{B}$.

Reducciones



Proceso de Traducción

Reducciones

Es fácil ver que nuestra relación de reducción entre problemas es reflexiva (reducir $\mathscr{A}$ a $\mathscr{A}$ es no hacer nada) y transitiva (si $\mathscr{A} \leq \mathscr{B}$ y $\mathscr{B} \leq \mathscr{C}$ podemos traducir una instancia de $\mathscr{A}$ en una instancia de $\mathscr{C}$ con la misma respuesta en dos pasos). La relación de reducción no es antisimétrica ($\mathscr{A} \leq \mathscr{B}$ y $\mathscr{B} \leq \mathscr{A}$ no permiten concluir que los problemas $\mathscr{A}$ y $\mathscr{B}$ son el mismo) con lo que no es una relación de orden, pero es similar.

Uso de reducciones

Usaremos reducciones para demostrar por contradicción que ciertos problemas no pueden resolverse: si $\mathscr{A}$ es imposible de resolver, y $\mathscr{A}$ se reduce a $\mathscr{B}$, $\mathscr{B}$ tampoco puede resolverse. Esto coincide con la interpretación intuitiva de $\mathscr{A} \leq \mathscr{B}$ como «$\mathscr{A}$ es más fácil que $\mathscr{B}$» (que resulta un tanto a contrapelo, comúnmente usamos reducciones para transformar problemas desconocidos en problemas familiares).

Es imposible determinar si se llama una función

Como ejemplo usaremos la estrategia esbozada para demostrar que es imposible determinar si un programa, al leer datos dados, alguna vez llama a la función f.

Es imposible determinar si se llama una función

Reduciremos el problema «¡Hola, mundo!» al problema «¿Se llama f?». Dado un programa P que puede o no escribir «¡Hola, mundo!» al leer datos D, hacemos las siguientes modificaciones:

- Para evitar accidentes, renombramos todos los identificadores propios del programa P anteponiéndoles hola_. Si hay alguna función f, pasa a llamarse hola_f.
- Reemplazamos todas las funciones que producen salida (printf (3), putchar(3), . . . , write (2)) por nuestras propias versiones que acumulan lo escrito en memoria, y al ser llamadas verifican si lo ya escrito es «¡Hola, mundo!». De ser así, invocan a f.

Estas modificaciones son rutina, con algo de paciencia escribimos un *script* que automatice este proceso.

Es imposible determinar si se llama una función

Llamemos P' al programa resultante. Vemos que P' llama a f exactamente si P escribe «¡Hola, mundo!». Si pudiéramos determinar si P' llama a f, estaríamos en condiciones de resolver el problema de «¡Hola, mundo!», cosa que sabemos imposible.

Otros problemas imposibles

Muchos otros problemas sobre el comportamiento de programas son insolubles. Ejemplos incluyen:

- ¿Las funciones f y g dan siempre el mismo resultado? (Relevante en correctitud de programas: f es una versión que se sabe correcta, pero ineficiente; g es nuestra versión optimizada.)
- ¿Se usa alguna vez la variable x (o se le asigna alguna vez un valor, o toma alguna vez el valor 42, su valor es 1 en este punto del programa, u otras)? (Detectar errores u oportunidades de mejora del código en un compilador.)
- ¿Entra en un ciclo infinito este programa con algún dato?

Otros problemas imposibles

La estrategia en todos estos casos es similar a la esbozada para el problema «¿Se llama f?».

Note que hemos demostrado que no hay forma de resolver *todas* las instancias de los problemas dados. Esto no es lo mismo que decir que *ninguna* puede resolverse. Por ejemplo, es trivial ver que nuestro programa clásico escribe «¡Hola, mundo!», en muchos otros casos podemos demostrar que no hay forma que escriba tal cosa.

Al programar, debemos tener cuidado de movernos en el ámbito de programas que sabemos correctos. Determinar en forma independiente si el programa es correcto resulta imposible en general.

Aplicaciones

Los problemas respecto de programas mencionados (y muchos más) no pueden resolverse en general, pero pueden resolverse en muchos casos. Soluciones parciales (particularmente algoritmos rápidos, aun si no son todo lo precisos posible) son de interés al analizar programas, como lo hace un compilador para buscar oportunidades de mejora (eliminar variables que no se referencian, omitir cálculos cuyos valores no se usan, simplificar expresiones en que se conocen los valores de algunos operandos, reemplazar operaciones por otras equivalentes menos costosas).

Al solicitar «optimización» del compilador (realmente «haga un esfuerzo por generar código mejor») analiza el código en forma más extensa, con lo que probablemente detecte más construcciones dudosas que pueden ser síntoma de errores.

Resumen

- Mediante un simple ejemplo demostramos que ciertos problemas de interés no pueden ser resueltos por ningún algoritmo.
- Lamentablemente, muchos problemas de inmenso interés práctico caen en esta categoría.
- Que un problema no pueda resolverse en *todos* los casos no significa que no hayan casos especiales en que sí puede resolverse.
- Introdujimos el concepto de *reducción de problemas*, y lo usamos (por contradicción) para demostrar insolubles problemas adicionales de mayor interés que «¡Hola, mundo!».