4 Formalizaciones - Un Panorama II

José de Jesús Lavalle Martínez

Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación
Computabilidad CCOS 257

Contenido

Motivación

Para una segunda formalización del concepto de calculabilidad, definiremos cierta clase de funciones parciales sobre $\mathbb{N}$ como la clase más pequeña que contiene ciertas funciones iniciales y es cerrada bajo ciertas construcciones.

Recursividad primitiva y búsqueda I

Para las funciones iniciales, tomamos las siguientes funciones totales muy simples:

- Las funciones *cero*, esto es, las funciones constantes f definidas por la ecuación:

  $$f(x_1, \ldots, x_k) = 0.$$

  existe una de tales funciones para cada k.

Recursividad primitiva y búsqueda I

Para las funciones iniciales, tomamos las siguientes funciones totales muy simples:

- Las funciones *cero*, esto es, las funciones constantes f definidas por la ecuación:

  $$f(x_1, \ldots, x_k) = 0.$$

  existe una de tales funciones para cada k.
- La función *sucesor* S, definida por la ecuación:

  $$S(x) = x + 1.$$

Recursividad primitiva y búsqueda I

Para las funciones iniciales, tomamos las siguientes funciones totales muy simples:

- Las funciones *cero*, esto es, las funciones constantes f definidas por la ecuación:

  $$f(x_1, \ldots, x_k) = 0.$$

  existe una de tales funciones para cada k.
- La función *sucesor* S, definida por la ecuación:

  $$S(x) = x + 1.$$

- Las funciones *proyección* I_n^k de k-dimensiones sobre la coordenada n-ésima,

  $$I_n^k(x_1, \ldots, x_k) = x_n,$$

  donde $1 \leq n \leq k$.

Recursividad primitiva y búsqueda II

- Queremos formar la cerradura de la clase de funciones iniciales bajo tres construcciones: composición, recursión primitiva y búsqueda.

Recursividad primitiva y búsqueda II

- Queremos formar la cerradura de la clase de funciones iniciales bajo tres construcciones: composición, recursión primitiva y búsqueda.
- Una función h de aridad-k se dice que se obtiene por composición de la función f de aridad-n y las funciones $g_1, \ldots, g_n$ de aridad-k si la ecuación

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

se cumple para todo $\vec{x}$. En el caso de funciones parciales, se entiende aquí que $h(\vec{x})$ está indefinida al menos que $g_1(\vec{x}), \ldots, g_n(\vec{x})$ estén todas definidas y que $\langle g_1(\vec{x}), \ldots, g_n(\vec{x})\rangle$ pertenezca al dominio de f.

Recursividad primitiva y búsqueda II

- Queremos formar la cerradura de la clase de funciones iniciales bajo tres construcciones: composición, recursión primitiva y búsqueda.
- Una función h de aridad-k se dice que se obtiene por composición de la función f de aridad-n y las funciones $g_1, \ldots, g_n$ de aridad-k si la ecuación

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

se cumple para todo $\vec{x}$. En el caso de funciones parciales, se entiende aquí que $h(\vec{x})$ está indefinida al menos que $g_1(\vec{x}), \ldots, g_n(\vec{x})$ estén todas definidas y que $\langle g_1(\vec{x}), \ldots, g_n(\vec{x})\rangle$ pertenezca al dominio de f.
- Una función h de aridad-$k+1$ se dice que se obtine por *recursión primitiva* de la función f de aridad-k y la función g de aridad-$(k+2)$ (donde $k > 0$) si el par de ecuaciones

$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, y+1) = g(h(\vec{x}, y), \vec{x}, y)$$

se cumple para todo $\vec{x}$ y y.

Recursividad primitiva y búsqueda III

- Nuevamente, en el caso de funciones parciales, se entiende que $h(\vec{x}, y+1)$ está indefinida al menos que $h(\vec{x}, y)$ esté definida y que $\langle h(\vec{x}, y), \vec{x}, y\rangle$ esté en el dominio de g.

Recursividad primitiva y búsqueda III

- Nuevamente, en el caso de funciones parciales, se entiende que $h(\vec{x}, y+1)$ está indefinida al menos que $h(\vec{x}, y)$ esté definida y que $\langle h(\vec{x}, y), \vec{x}, y\rangle$ esté en el dominio de g.
- Observe que en esta situación, conocer las dos funciones f y g determina completamente la función h. Más formalmente, si tanto h_1 como h_2 se obtinen por recursión primitiva de f y g, entonces para cada $\vec{x}$, podemos mostrar por inducción sobre y que $h_1(\vec{x}, y) = h_2(\vec{x}, y)$.

Recursividad primitiva y búsqueda III

- Nuevamente, en el caso de funciones parciales, se entiende que $h(\vec{x}, y+1)$ está indefinida al menos que $h(\vec{x}, y)$ esté definida y que $\langle h(\vec{x}, y), \vec{x}, y\rangle$ esté en el dominio de g.
- Observe que en esta situación, conocer las dos funciones f y g determina completamente la función h. Más formalmente, si tanto h_1 como h_2 se obtinen por recursión primitiva de f y g, entonces para cada $\vec{x}$, podemos mostrar por inducción sobre y que $h_1(\vec{x}, y) = h_2(\vec{x}, y)$.
- Para el caso $k = 0$, la función h de aridad-uno se obtiene por recursión primitiva de la función g de aridad-dos usando el número m si el par de ecuaciones

$$h(0) = m$$
$$h(y+1) = g(h(y), y)$$

se cumplen para todo y.

Recursividad primitiva y búsqueda IV

- Posponiendo el asunto de la búsqueda, definimos a una función como *recursiva primitiva* si se puede construir de las funciones cero, sucesor y proyección mediante el uso de composición y recursión primitiva.

Recursividad primitiva y búsqueda IV

- En otras palabras, la clase de funciones recursivas primitivas es la clase más pequeña que incluye nuestras funciones iniciales y es cerrada bajo composición y recursión primitiva.

Recursividad primitiva y búsqueda IV

- Aquí decir que una clase $\mathcal{C}$ es "cerrada" bajo composición y recursión primitiva significa que siempre que una función f se obtiene por composición de funciones en $\mathcal{C}$ o se obtiene por recursión primitiva de funciones en $\mathcal{C}$, entonces f misma pertenece a $\mathcal{C}$.

Recursividad primitiva y búsqueda IV

- Claramente todas las funciones recursivas primitivas son totales. Esto es porque todas las funciones inciales son totales, la composición de funciones totales es total, y una función obtenida por recursión primitiva a partir de funciones totales será total.

Recursividad primitiva y búsqueda V

- Decimos que una relación R sobre $\mathbb{N}$ de aridad-k es recursiva primitiva si su función característica es recursiva primitiva.

Recursividad primitiva y búsqueda V

- Luego uno puede mostrar que una gran cantidad de las funciones comunes sobre $\mathbb{N}$ son recursivas primitivas: adición, multiplicación, . . . , la función cuyo valor en m es el $(m+1)$-ésimo primo, ...

Recursividad primitiva y búsqueda V

- En el capítulo 2 emprenderemos el proyecto de mostrar que muchas funciones son recursivas primitivas.

Recursividad primitiva y búsqueda V

- Por un lado, parece claro que toda función recursiva primitiva debe considerarse como calculable efectivamente.

Recursividad primitiva y búsqueda V

- Las funciones iniciales son muy fáciles. La composición no presenta grandes obstáculos.

Recursividad primitiva y búsqueda V

- Siempre que h se obtiene por recursión primitiva de f y g calculables efectivamente, entonces vemos que podemos encontrar efectivamente $h(\vec{x}, 99)$, encontrando primero $h(\vec{x}, 0), h(\vec{x}, 1), \ldots, h(\vec{x}, 98)$.)

Recursividad primitiva y búsqueda V

- Por otro lado, es posible que la clase de funciones recursivas primitivas no pueda incluir a todas las funciones calculables totales ya que podemos "diagonalizar" la clase.

Recursividad primitiva y búsqueda VI

- Esto es, mediante un indexado apropiado del "árbol familiar" de las funciones recursivas primitivas, podemos hacer una lista $f_0, f_1, f_2, \ldots$ de todas las funciones recursivas primitivas de aridad-uno. Luego considere la función diagonal $d(x) = f_x(x) + 1$.

Recursividad primitiva y búsqueda VI

- Esto es, mediante un indexado apropiado del "árbol familiar" de las funciones recursivas primitivas, podemos hacer una lista $f_0, f_1, f_2, \ldots$ de todas las funciones recursivas primitivas de aridad-uno. Luego considere la función diagonal $d(x) = f_x(x) + 1$.
- Entonces d no puede ser recursiva primitiva; difiere de cada f_x en x. Sin embargo, si hicimos nuestra lista muy pulcramente, la función d será calculable efectivamente.

Recursividad primitiva y búsqueda VI

- Esto es, mediante un indexado apropiado del "árbol familiar" de las funciones recursivas primitivas, podemos hacer una lista $f_0, f_1, f_2, \ldots$ de todas las funciones recursivas primitivas de aridad-uno. Luego considere la función diagonal $d(x) = f_x(x) + 1$.
- Entonces d no puede ser recursiva primitiva; difiere de cada f_x en x. Sin embargo, si hicimos nuestra lista muy pulcramente, la función d será calculable efectivamente.
- La conclusión es que la clase de funciones recursivas primitivas es una clase extensa pero es una subclase de las funciones calculables totales.

Recursividad primitiva y búsqueda VII

- Luego, decimos que una función h de aridad-k se obtiene de una función g de aridad-$(k+1)$ mediante *búsqueda*, y escribimos

$$h(\vec{x}) = \mu\, y[g(\vec{x}, y) = 0]$$

si para cada $\vec{x}$, el valor $h(\vec{x})$ o es el número y tal que $g(\vec{x}, y) = 0$ y $g(\vec{x}, y)$ está definida y no es cero para todo $s < y$, si tal número y existe, sino está indefinida, si tal número y no existe.

Recursividad primitiva y búsqueda VII

- Luego, decimos que una función h de aridad-k se obtiene de una función g de aridad-$(k+1)$ mediante *búsqueda*, y escribimos

$$h(\vec{x}) = \mu\, y[g(\vec{x}, y) = 0]$$

  si para cada $\vec{x}$, el valor $h(\vec{x})$ o es el número y tal que $g(\vec{x}, y) = 0$ y $g(\vec{x}, y)$ está definida y no es cero para todo $s < y$, si tal número y existe, sino está indefinida, si tal número y no existe.
- La idea detrás de este "operador-μ" es la de buscar por el número más pequeño y tal que es la solución a una ecuación, probando sucesivamente con $y = 0, 1, \ldots$.

Recursividad primitiva y búsqueda VIII

- Obtenemos las funciones *recursivas generales* agregando búsqueda a nuestros métodos de cerradura.

Recursividad primitiva y búsqueda VIII

- Esto es, una función parcial es recursiva general si se puede construir a partir de las funciones cero, sucesor y proyección, usando composición, recursión primitiva y búsqueda (es decir, el operador-μ).

Recursividad primitiva y búsqueda VIII

- La clase de funciones parciales recursivas generales sobre $\mathbb{N}$ es (como lo demostró Turing) exactamente la misma como la clase de funciones parciales computables a la Turing.

Recursividad primitiva y búsqueda VIII

- Este es un resultado muy llamativo, a la luz de las maneras muy distintas en las que las dos definiciones fueron formuladas.

Recursividad primitiva y búsqueda VIII

- Las máquinas de Turing parecerían, a primera vista, tener poco que ver con recursión primitiva y búsqueda. Y aún así, obtenemos exactamente las mismas funciones parciales a partir de los dos enfoques.

Recursividad primitiva y búsqueda IX

- Y la tesis de Church, por tanto, tiene la formulación equivalente a que el concepto de función general recursiva es la formalización correcta del concepto informal de calculabilidad efectiva.

Recursividad primitiva y búsqueda IX

- ¿Qué si tratamos de “diagonalizar” la clase de funciones recursivas generales, como lo hicimos con las funciones recursivas primitivas?

Recursividad primitiva y búsqueda IX

- Como argumentaremos más tarde, podemos hacer nuevamente una lista ordenada $\phi_0, \phi_1, \phi_2, \ldots$ de todas las funciones parciales reursivas generales.

Recursividad primitiva y búsqueda IX

- Podemos definir la función diagonal $d(x) = \phi_x(x) + 1$. Pero en esta ecuación, $d(x)$ está indefinida al menos que $\phi_x(x)$ esté definida.

Recursividad primitiva y búsqueda IX

- La función diagonal d está en realidad entre las funciones parciales recursivas generales, y así es ϕ_k para algún k, pero $d(k)$ debe estar indefinida.

Recursividad primitiva y búsqueda IX

- Ninguna contradicción resulta.

Recursividad primitiva y búsqueda X

- El uso de la palabra “recursiva” en el contexto de las funciones recursivas primitivas es enteramente razonable.

Recursividad primitiva y búsqueda X

- Gödel, escribiendo en Alemán, había usado simplemente “rekursiv” para las funciones recursivas primitivas.

Recursividad primitiva y búsqueda X

- Fue Rózsa Péter quien introdujo el término “recursiva primitiva”.

Recursividad primitiva y búsqueda X

- Pero la clase de funciones recursivas generales tiene -como esta sección muestra- otras caracterizaciones en las que la *recursión* juega un papel nada obvio.

Recursividad primitiva y búsqueda XI

- Esto nos guía a la pregunta: ¿Cómo llamar a esta clase de funciones?

Recursividad primitiva y búsqueda XI

- Tener dos nombres ("computable a la Turing" y "recursiva general") es una vergüenza de ricos, y la situación sólo empeorará.

Recursividad primitiva y búsqueda XI

- Históricamente, el nombre “funciones recursivas parciales” ganó.

Recursividad primitiva y búsqueda XI

- Y las relaciones sobre $\mathbb{N}$ se dicen *recursivas* si sus funciones características pertenecen a esa clase.

Recursividad primitiva y búsqueda XI

- El estudio de esas funciones por años fue llamada “teoría de funciones recursivas”, y luego “teoría de recursión”.

Recursividad primitiva y búsqueda XI

- Pero esto fue más un accidente histórico que una elección razonada.

Recursividad primitiva y búsqueda XI

- No obstante, la terminología se ha vuelto estándar.

Recursividad primitiva y búsqueda XI

- Pero ahora se está haciendo un esfuerzo para cambiar lo que ha sido la terminología estándar.

Recursividad primitiva y búsqueda XI

- En consecuencia, este libro, *Teoría de la Computabilidad*, habla de funciones parciales *computables*.

Recursividad primitiva y búsqueda XI

- Y le llamaremos a una relación *computable* si su función característica es una función computable.

Recursividad primitiva y búsqueda XI

- Así, el concepto de relación computable corresponde a la noción informal de relación decidible.

Recursividad primitiva y búsqueda XI

- El manuscrito de este libro ha, no obstante, sido preparado con macros TEX que facilitarían un cambio rápido de terminología.

Recursividad primitiva y búsqueda XI

- En cualquier caso, categóricamente existe la necesidad de tener adjetivos separados para el concepto informal (aquí “calculable” se usa para funciones, y “decidible” para relaciones) y el concepto definido formalmente (aquí “computable”).

Programas Loop y While I

- La idea detrás del concepto de funciones calculables efectivas es que uno debería poder dar instrucciones explícitas (un programa) para calcular dicha función.

Programas Loop y While I

- ¿Qué lenguaje de programación sería adecuado aquí?

Programas Loop y While I

- En realidad, cualquiera de los lenguajes de programación comúnmente utilizados sería suficiente, si estuviera libre de ciertas limitaciones prácticas, como el tamaño del número denotado por una variable.

Programas Loop y While I

- Damos aquí un lenguaje de programación simple con la propiedad de que las funciones programables son exactamente las funciones parciales computables sobre $\mathbb{N}$.

Programas Loop y While II

- Las variables del lenguaje son $X_0, X_1, X_2, \ldots$.

Programas Loop y While II

- Aunque hay infinitas variables en el lenguaje, cualquier programa, al ser una cadena finita de comandos, sólo puede tener un número finito de estas variables.

Programas Loop y While II

- Si queremos que el lenguaje esté formado por palabras sobre un alfabeto finito, podemos reemplazar X_3, digamos, por X'''.

Programas Loop y While II

- Al ejecutar un programa, a cada variable del programa se le asigna un número natural. No hay límite sobre cuán grande puede ser este número.

Programas Loop y While II

- Inicialmente, algunas de las variables contendrán la entrada a la función; el lenguaje no tiene comandos de “entrada”.

Programas Loop y While II

- De manera similar, el lenguaje no tiene comandos de “salida”; cuando (y si) el programa se detiene, el valor de X_0 debe ser el valor de la función.

Programas Loop y While III

- Los comandos del lenguaje son de cinco tipos:

Programas Loop y While III

- Los comandos del lenguaje son de cinco tipos:
  - $X_n \leftarrow 0$. Este es el comando de *borrado*; su efecto es asignar el valor 0 a X_n.

Programas Loop y While III

- Los comandos del lenguaje son de cinco tipos:
  - $X_n \leftarrow X_n + 1$. Este es el comando de *incremento*; su efecto es aumentar en uno el valor asignado a X_n.

Programas Loop y While III

- Los comandos del lenguaje son de cinco tipos:
  - $X_n \leftarrow X_m$. Este es el comando de *copia*; su efecto es justo lo que sugiere el nombre; en particular, deja el valor de X_m sin cambios.

Programas Loop y While III

- Los comandos del lenguaje son de cinco tipos:

  - loop X_n y endloop X_n. Estos son los comandos *loop* y deben usarse en pares. Es decir, si $\mathcal{P}$ es un programa (una cadena de comandos sintácticamente correcta), entonces también lo es la cadena:

$$\begin{array}{l} \text{loop } X_n \\ \quad \mathcal{P} \\ \text{endloop } X_n \end{array}$$

Programas Loop y While IV

- Lo que este programa significa es que $\mathcal{P}$ debe ejecutarse un cierto número k de veces.

Programas Loop y While IV

- Y ese número k es el valor inicial de X_n, el valor asignado a X_n antes de comenzar a ejecutar $\mathcal{P}$.

Programas Loop y While IV

- Posiblemente $\mathcal{P}$ cambie el valor de X_n; esto no tiene ningún efecto sobre k.

Programas Loop y While IV

- Si $k = 0$, entonces esta cadena no hace nada.

Programas Loop y While V

- El quinto comando es:

Programas Loop y While V

- El quinto comando es:
  - while $X_n \neq 0$ y endwhile $X_n \neq 0$. Estos son los comandos *while*; nuevamente, deben usarse en pares, como los comandos de bucle. Pero hay una diferencia. El programa

$$\begin{array}{l} \text{while } X_n \neq 0 \\ \quad \mathcal{P} \\ \text{endwhile } X_n \neq 0 \end{array}$$

también ejecuta el programa $\mathcal{P}$ un número k de veces.

Programas Loop y While V

- El quinto comando es:
  - while $X_n \neq 0$ y endwhile $X_n \neq 0$. Estos son los comandos *while*; nuevamente, deben usarse en pares, como los comandos de bucle. Pero hay una diferencia. El programa

$$\begin{array}{l} \text{while } X_n \neq 0 \\ \quad \mathcal{P} \\ \text{endwhile } X_n \neq 0 \end{array}$$

    también ejecuta el programa $\mathcal{P}$ un número k de veces.
  - Pero ahora k no está determinado de antemano; importa mucho cómo cambia $\mathcal{P}$ el valor de X_n. El número k es el número menor (si lo hay) tal que $\mathcal{P}$ se ejecutará hasta que a X_n se le asigne el valor 0. El programa se ejecutará para siempre si no existe tal k.

Programas Loop y While VI

- Y esos son los únicos comandos.

Programas Loop y While VI

- Un programa *while* es una secuencia de comandos, sujeta únicamente al requisito de que el bucle y los comandos while se utilicen en pares, como se ilustra.

Programas Loop y While VI

- Claramente, este lenguaje de programación es lo suficientemente simple como para ser simulado por cualquiera de los lenguajes de programación comunes si ignoramos los problemas de desbordamiento.

Programas Loop y While VI

- Un programa *loop* es un programa while sin comandos while; es decir, sólo tiene comandos de borrado, incremento, copia y loop.

Programas Loop y While VI

- Tenga en cuenta la propiedad importante: un programa loop *siempre se detiene*, pase lo que pase.

Programas Loop y While VI

- Pero es fácil hacer un programa while que nunca se detenga.

Programas Loop y While VII

- Decimos que una función parcial f de aridad k sobre $\mathbb{N}$ es *computable-while* si existe un programa while $\mathcal{P}$ que, siempre que comience con una tupla $\vec{x}$ de aridad k asignada a las variables $X_1, \ldots, X_k$ y 0 asignado al resto de variables, se comporta de la siguiente manera:

Programas Loop y While VII

- Decimos que una función parcial f de aridad k sobre $\mathbb{N}$ es *computable-while* si existe un programa while $\mathcal{P}$ que, siempre que comience con una tupla $\vec{x}$ de aridad k asignada a las variables $X_1, \ldots, X_k$ y 0 asignado al resto de variables, se comporta de la siguiente manera:
  - Si $f(\vec{x})$ está definida, entonces el programa eventualmente se detiene, y X_0 contiene el valor de $f(\vec{x})$.

Programas Loop y While VII

- Decimos que una función parcial f de aridad k sobre $\mathbb{N}$ es *computable-while* si existe un programa while $\mathcal{P}$ que, siempre que comience con una tupla $\vec{x}$ de aridad k asignada a las variables $X_1, \ldots, X_k$ y 0 asignado al resto de variables, se comporta de la siguiente manera:
  - Si $f(\vec{x})$ está definida, entonces el programa eventualmente se detiene, y X_0 contiene el valor de $f(\vec{x})$.
  - Si $f(\vec{x})$ no está definida, entonces el programa nunca se detiene.

Programas Loop y While VII

- Decimos que una función parcial f de aridad k sobre $\mathbb{N}$ es *computable-while* si existe un programa while $\mathcal{P}$ que, siempre que comience con una tupla $\vec{x}$ de aridad k asignada a las variables $X_1, \ldots, X_k$ y 0 asignado al resto de variables, se comporta de la siguiente manera:
  - Si $f(\vec{x})$ está definida, entonces el programa eventualmente se detiene, y X_0 contiene el valor de $f(\vec{x})$.
  - Si $f(\vec{x})$ no está definida, entonces el programa nunca se detiene.
- Las funciones *computables-loop* se definen de forma análoga.

Programas Loop y While VII

- Decimos que una función parcial f de aridad k sobre $\mathbb{N}$ es *computable-while* si existe un programa while $\mathcal{P}$ que, siempre que comience con una tupla $\vec{x}$ de aridad k asignada a las variables $X_1, \ldots, X_k$ y 0 asignado al resto de variables, se comporta de la siguiente manera:
  - Si $f(\vec{x})$ está definida, entonces el programa eventualmente se detiene, y X_0 contiene el valor de $f(\vec{x})$.
  - Si $f(\vec{x})$ no está definida, entonces el programa nunca se detiene.
- Las funciones *computables-loop* se definen de forma análoga.
- Pero existe la diferencia de que cualquier función computable-loop es total.

Programas Loop y While VIII

Teorema 1

(a) Una *función* sobre $\mathbb{N}$ es *computable-loop* si y sólo si es *recursiva primitiva*.

(b) Una *función parcial* sobre $\mathbb{N}$ es *computable-while* si y sólo si es *recursiva general*.

Programas Loop y While IX

- La prueba en una dirección, para demostrar que toda función recursiva primitiva es computable-loop, implica una serie de ejercicios de programación.

Programas Loop y While IX

- La prueba en la otra dirección implica codificar el estado de un programa $\mathcal{P}$ sobre la entrada $\vec{x}$ después de t pasos, y mostrar que existen funciones recursivas primitivas que nos permiten determinar el estado después de $t+1$ pasos y el estado terminal.

Programas Loop y While IX

- Debido a que la clase de funciones parciales recursivas generales coincide con la clase de funciones parciales computables de Turing, podemos concluir del teorema anterior que la computabilidad while coincide con la computabilidad de Turing.