La Programación Considerada como una Actividad Humana

E. W. Dijkstra, 1965*

Introducción

A manera de introducción, me gustaría comenzar esta charla con una historia y una cita.

La historia es acerca del físico Ludwig Boltzmann, quien no tenía problemas para alcanzar sus objetivos a través de lentos cálculos. Un día alguien le reclamó acerca de la poca elegancia de sus métodos, ante lo cual Boltzmann defendió su forma de trabajar diciendo que "la elegancia era preocupación de sastres y zapateros", implicando con esto que se negaba a ocuparse de ello.

En constraste a esto, me gustaría citar a otro famoso científico del siglo XIX, George Boole. En la mitad de su libro, *Investigación sobre las leyes del pensamiento*, en un capítulo titulado "Sobre las condiciones del método perfecto", escribe: "No hablo aquí sólo de la perfección que consiste en el poder, sino también en aquella que se funda en la concepción de lo que es adecuado y bello. Es probable que un análisis cuidadoso de este asunto nos conduzca a conclusiones tales como, por ejemplo, que un método perfecto no debiera ser sólo eficiente respecto de los objetivos para los cuales es diseñado, sino debiera manifestar en todas sus partes y procesos una cierta unidad y armonía." Una diferencia en actitud que es difícil pasar por alto.

Nuestra asociación inconsciente entre elegancia y lujo puede que sea uno de los orígenes de la no inusual y tácita suposición que ser elegante tiene un costo. El mostrar que también ser elegante trae beneficios es unos de mis principales propósitos. Ello nos dará una comprensión más clara de la verdadera naturaleza de la calidad de los programas y de la forma en que ellos se expresan, esto es, el lenguaje de programación. De este *insight* trataremos

*En Proceedings of the 1965 IFIP Congress, Amsterdam, 1965, North Holland Publ. Co., pp. 213-217. Tomado de Classics in Software Engineering, Ed. E. N. Yourdon, Yourdon Press, 1979. Trad. C. Gutiérrez, 2001.

de derivar ciertas claves acerca de cuáles funcionalidades de los lenguajes de programación son más deseables. Finalmente, esperamos convencerlos a ustedes que los diferentes objetivos son menos conflictivos entre sí que lo que aparecen a primera vista.

Sobre la calidad de los resultados

Aún bajo la suposición de máquinas que trabajan sin errores nos debieramos preguntar: "Cuando un computador automático produce resultados, ¿porqué confiamos en ellos, si lo hacemos? y a continuación: "¿Que medidas podemos tomar para incrementar nuestra confianza en que los resultados producidos son efectivamente los resultados esperados?"

El cuan urgente es la primera pregunta puede se ilustrado por un ejemplo sencillo, algo simplificado. Supongamos que un matemático interesado en teoría de números tiene a su disposición un programa para factorizar números. Este proceso puede terminar de una de dos formas: o da la factorización del número dado o responde que el número es primo. Supongamos ahora que nuestro matemático desea someter a este proceso, digamos, un número de 20 decimales, teniendo fuertes razones para suponer que el es un número primo. Si la máquina confirma esta esperanza, estará contento; si encuentra una factorización, el matemático pudiera sentirse defraudado pues su intuición lo engañó una vez más, pero, en la duda, puede tomar una calculadora y multiplicar los factores producidos para verificar si el producto reproduce el número original. Esta situación cambia drásticamente, sin embargo, si el espera que el número dado no sea primo: si la máquina ahora produce factores sus previsiones se verán confirmadas y más aún puede verificar el resultado multiplicado. Si, por el contrario, la máquina retorna con la respuesta que el número es, contrariamente a sus espectativas y cálidos deseos, un número primo, ¿quien sobre la tierra debiera creer esto?

Nuestro ejemplo muestra que aún en problemas completamente discretos la computación de un resultado no es un trabajo bien definido, bien definido en el sentido que uno puede decir'"Lo hice", si poner atención al poder de convicción del resultado, es decir, a su *calidad*.

La situación del programador es bastante análoga a la del matemático puro, que desarrolla una teoría y demuestra su resultado. Por un largo tiempo los matemáticos puros han pensado —y algunos aún lo piensan— que un teorema puede ser completamente demostrado, que el problema de si una supuesta demostración para un teorema es suficiente o no, admite una respuesta absoluta "si" o "no". Pero esto es una ilusión, puesto que apenas uno piensa que uno ha demostrado algo, todavía tiene la tarea de demostrar que la primera demostración no contenía errores, y así sucesivamente, ¡*ad infinitum*. Uno nunca puede garantizar que una demostración es correcta; lo mejor que uno puede decir es: "No he descubierto ningún error". A veces nos adulamos con la idea de dar demostraciones detalladas, pero de hecho no hacemos más que hacer la correctitud de nuestra demostración plausible. Tan extremadamente plausible, que la analogía puede servir como una gran fuente de inspiración.

A contrapelo de sus deficiencias, el razonamiento matemático presenta un modelo destacado acerca de cómo capturar estructuras extremadamente complicadas con un cerebro de capacidad limitada. Y parece que vale la pena investigar hasta que límites estos métodos probados pueden ser transplantados al arte del uso del computador. En el diseño de lenguajes de programación uno puede dejarse guiar primariamente por la consideración de "lo que la máquina puede hacer". Considerando, sin embargo, que los lenguajes de programación son el puente entre el usuario y la máquina —que puede ser considerada, en efecto, como su herramienta— surge como importante considerar "lo que el Hombre puede pensar". Es en este espíritu que continuaremos nuestras investigaciones.

Sobre la estructura de los programas convincentes

La técnica para dominar la complejidad ha sido conocida desde tiempos antiguos" *Divide et impera* (Divide y conquista). La analogía entre la construcción de demostraciones y la construcción de programas es, de nuevo, sorprendente. En ambos casos los puntos de partida disponibles están dados (axiomas y teoría existente versus primitivas y bibliotecas de programas disponibles); en ambos casos el objetivo está dado (el teorema a ser demostrado versus el rendimiento deseado); en ambos casos la complejidad es abordada por división en partes (lemas versus subprogramas y procedimientos).

Supongo que el genio del programador está a la altura de las dificultades de su problema y supongo que ha llegado a una adecuada subdivisión de la tarea. Entonces procede de la forma usual a través de los siguientes pasos:

- Hace las especificaciones completas de las partes individuales
- Considera que el problema total está resuelto cuando tiene a su disposición las partes del programa que satisfacen las correspondientes especificaciones
- Construye las partes individuales, satisfaciendo las especificaciones, pero en forma independiente una de otra y del contexto futuro en el cual ellas serán usadas.

Obviamente, la construcción de una tal parte individual puede ser de nuevo una tarea de tal complejidad, que en esta parte del trabajo, una nueva subdivisión sea requerida.

Algunas personas pudieran pensar que la técnica de la disección recién expuesta es una forma algo indirecta y tortuosa de conseguir nuestros objetivos. Mis propios sentimientos son mejor descritos, quizás, diciendo que estoy perfectamente consciente de que no hay Camino Real a las Matemáticas, o en otras palabras, que tengo sólo una pequeña cabeza y debo vivir con ella. Por consiguiente, veo que la técnica de disección es uno de los patrones relativamente básico del entendimiento humano y pienso que es valioso intentar crear circunstancias en el cual pueda ser aplicado fructíferamente.

La suposición que el programador ha hecho una adecuada subdivisión encuentra su contraparte en la posibilidad de ejecutar las dos primeros etapas: la especificación de las partes y la verificación que ellas en conjunto hacen el trabajo. Aquí la elegancia, la precisión, la claridad y una completa comprensión del problema a resolver son prerequisitos. Pero la técnica de disección descansa en algo menos visible, a saber, en lo que yo llamaría "El principio de no interferencia". En la segunda etapa mencionada más arriba se supone que la corrección del trabajo como un todo puede establecerse tomando de las partes en consideración su especificación exterior solamente, y no los detalles de su construcción interior. En la tercera etapa este principio de no interferencia surge de nuevo: aquí se supone que las partes individuales pueden ser concebidas y construídas independientemente una de otra.

Este es quizás el momento para mencionar que, si yo interpreto correctamente los signos de las actitudes actuales hacia los problemas de la definición de lenguajes, en algunos enfoques más formalistas la corrección de la técnica de disección se pone en duda. Sus sostenedores argumentan asi: cada vez que usted presenta un mecanismo como una definición en dos etapas, primero, lo que debiera hacer, esto es, su especificación, y segundo, como trabaja, usted en el mejor de los casos ha dicho la misma cosa dos veces, pero con toda probabilidad se contradijo a sí mismo. Y hablando estad'ısticamente, siento decirlo, esta última observación es un punto fuerte. La única forma clara para la definición de un lenguaje, argumentan, es a través de la definición de los mecanismos, puesto que lo que ellos harán se seguirá de esa definición. Mi pregunta': "Como se sigue esto?" es dejada hábilmente sin respuesta y temo que su descuido de la sutil, aunque a veces formidable, diferencia entre los conceptos *definido* y *conocido* hará que sus esfuerzos sean ejercicio intelectual que lleva a otra avenida ciega.

Después de esta excursión retornamos a la programación misma. Todos aquellos familiarizados con ALGOL 60 coincidirán que su concepto de procedimiento satisface en medida razonable nuestros requerimientos de no interferencia, tanto en sus propiedades estáticas, por ejemplo en la libertad para la elección de identificadores locales, como en sus propiedades dinámicas, por ejemplo la posibilidad de llamar a un procedimiento, directa o indirectamente, desde dentro de sí mismo.

Otro ejemplo sorprendente de incremento de la claridad a través de la no interferencia, garantizada por la estructura, se presenta en todos los lenguajes de programación en los cuales se permite expresiones algebraicas. La evaluación de tales expresiones con una máquina secuencial que tiene una unidad aritmética de complejidad limitada implicará el uso de memoria temporal para los resultados intermedios. Su anonimidad en el lenguaje fuente garantiza la imposibilidad de que uno de esos resultados intermedios sea eliminado inadvertidamente antes de que sea usado, como podría ocurrir si el proceso computacional fuese descrito en un código de máquina de tipo Von Neumann.

Una comparación de alternativas

Una amplia comparación entre el código de máquina de tipo Von Neumann —bien conocido por su falta de claridad— y diferentes tipos de lenguajes algorítmicos puede ser relevante.

En todos los casos, la ejecución de un programa consiste en la confrontación repetida de dos flujos de información, uno (digamos *el programa*) constante en el tiempo, el otro (digamos *los datos*) variando. Por muchos años se ha pensado como una de las virtudes esenciales del código tipo Von Neumann el que el programa pudiera modificar sus propias instrucciones. Entretanto hemos descubierto que exactamente esta propiedad es la responsable en gran medida de la falta de claridad en los programas de código de máquina. Simultáneamente su indispensabilidad ha sido cuestionada: todos los compiladores algebraicos que conozco producen un programa objeto que permanece constante durante toda su fase de ejecución.

Estas observaciones nos llevan a considerar el estatus de la información variable. Confinemos primero nuestra atención a los lenguajes de programación sin asignación y sin *goto*. Suponiendo que el espectro de funciones admisibles es suficientemente amplio y el concepto de condicional esté entre las primitivas disponibles, uno podría escribir la salida de cada programa como el valor de una función (recursiva) grande. Para una máquina secuencial esto puede traducirse en un programa objeto constante, en el cual en tiempo de ejecución se use una pila para llevar la cuenta de la actual jerarquía de llamados y de los valores de los actuales parámetros pasados a estos llamados.

A pesar de su elegancia, es posible hacer una seria objeción a tal lenguaje de programación. Aquí la información en la pila puede ser vista como objetos con tiempos de vida anidados y con un valor constante durante todo su tiempo de vida. Nunca (excepto en el incremento implícito del contador de orden que captura el progreso del tiempo) el valor de un objeto nombrado ya existente es reemplazado por otro valor. Como resultado, la única forma de guardar un resultado nuevo reciente es a través de ponerlo en la pila; no tenemos ninguna forma de expresar que un valor anterior ahora quedó obsoleto y por consiguiente su tiempo de vida será prolongado, aunque no tenga interés. Resumiendo: es elegante pero inadecuado. Una segunda objeción —que es probablemente una consecuencia directa de la primera— es que tales programas devienen después de un cierto grado de anidamiento (que se obtiene rápidamente) extraordinariamente difíciles de leer.

El remedio usual es la introducción combinada de la instrucción *goto* y de la instrucción de asignación. El *goto* nos permite con un salto hacia atrás repetir una pedazo de programa, mientras que la instrucción de asignación puede crear las diferencias necesarias en el estatus entre las sucesivas repeticiones.

Pero tengo razones para preguntar, si la instrucción *goto* como remedio no es peor que la enfermedad. Por ejemplo, dos administradores de departamentos de programación de diferentes países y diferente formación —uno principalmente científico, el otro principalmente comercial— me han comunicado independientemente y por su propia iniciativa, sus observaciones que la calidad de sus programadores era inversamente proporcional a la densidad de las instrucciones *goto* en sus programas. Esto ha sido un incentivo para tratar de eliminar la instrucción *goto*.

La idea es, que lo que conocemos como *transferencia de control*, es decir, el reemplazo del valor del contador de orden, es una operación a menudo incluída como parte de nociones más poderosas: menciono la transición a la siguiente instrucción, la llamada a procedimiento y retorno, las cláusulas condicionales y la instrucción *for*; y queda la pregunta si el programador no está algo dejado de lado al darle control separado sobre el.

He hecho varios experimentos de programación y comparado el texto de ALGOL con el texto que obtuve en versiones modificadas de ALGOL 60 en los cuales la instrucción *goto* fué abolida y la instrucción *for* —pomposa y sobreelaborada— fué reemplazada por una cláusula de repetición primitiva. Estas últimas versiones eran más difíciles de hacer: estamos tan familiarizados con el orden de salto, ¡que requiere algún esfuerzo olvidarse de él! En todos los casos experimentados, sin embargo, el programa sin las instrucciones *goto* resultó ser más corto y más claro.

El origen del incremento en la claridad es muy entendible. Como es bien sabido, no existe algoritmo para decidir si un programa dado termina o no. En otras palabras, cada programador que desea producir un programa sin errores debe al menos convencerse por inspección que su programa de hecho terminará. En un programa en el cual se ha hecho uso indiscriminado de la instrucción *goto*, este análisis será muy difícil debido a la gran variedad de formas en el cual el programa pudiera no terminar. Después de la abolición de la instrucción *goto* hay sólo dos formas en el cual el programa pudiera no terminar: o por una recursión infinita, es decir, a través de un mecanismo de procedimientos, o por la cláusula de repetición. Esto simplifica la inspección en gran medida.

La noción de repetición, tan fundamental en programación, tiene una otra consecuencia. No es raro que adentro de una sucesión de instrucciones a ser repetidas, ocurra una o más subexpresiones que no cambian su valor durante la repetición. Si tal sucesión se repite muchas veces, sería una lamentable pérdida de tiempo si la máquina tuviese que recomputar esos mismos valores una y otra vez. Una forma de evitar esto es delegar al ahora traductor optimizador el descubrimiento de tales subexpresiones constantes de forma que puede llevar el cómputo de esas expresiones fuera del bucle. Sin un traductor optimizador la solución obvia es invitar al programador a ser algo más explícito, y puede serlo introduciendo tantas variables adicionales como subexpresiones constantes existen adentro de la repetición y asignándoles estos valores a ellas antes de entrar en la repetición. Me gustaría enfatizar que ambas formas de escribir el programa son igualmente erróneas. En el primer caso, el traductor se enfrenta con el innecesario puzzle de descubrir las constantes; en el segundo caso hemos introducido una nueva variable, cuya única función es denotar un valor constante. La última observación muestra la forma de salir de esta dificultad: aparte de variables el programador debiera tener acceso a *constantes locales*, es decir, cantidades identificables con un tiempo de vida finito, durante los cuales tendrán un valor constante que ha sido definido en el momento de la introducción de la cantidad. Tales cantidades no son nuevas: los parámetros formales de procedimientos ya poseen esta propiedad. Lo anterior es un llamado a reconocer que el concepto de *constante local* tiene su propio derecho de existencia. Si yo estoy bien informado, esto ha sido ya reconocido en CPL, el lenguaje de programación diseñado en un esfuerzo conjunto en el Mathematical Laboratory de la Universidad de Cambridge, Inglaterra.

La doble ganancia en claridad

He discutido largamente que el poder de convencimiento de los resultados depende en gran medida de la claridad del programa, del grado en el cual el refleje la estructura del proceso a ser ejecutado. Para aquellos que se sienten más preocupados por la eficiencia medida en la crudas unidades de espacio de memoria y tiempo de máquina, quisiera señalarles que el incremento de la eficiencia siempre se reduce a la explotación de la estructura y para ellos me gustaría enfatizar que todas las propiedades estructurales mencionadas pueden ser usadas para incrementar la eficiencia de una implementación. Las resumiré brevemente.

La relación de tiempos de vida satisfecha por las cantidades locales de los procedimientos nos permite guardarlas en una pila, haciendo de esta forma un uso muy eficiente de la memoria disponible; el anonimato de los resultados intermedios nos permite minimizar dinámicamente referencias a memoria con la ayuda de un conjunto de acumuladores de pila controlado automáticamente; la constancia de textos de programa bajo ejecución es de gran ayuda en máquinas con diferentes niveles de memoria y reduce considerablemente la complejidad de un control avanzado; la cláusula de repetición simplifica la detección dinámica de bucles infinitos, y finalmente, la constante local es un buen candidato para una memoria de escritura lenta y lectura rápida (*write-slow-read-fast*), en caso de estar disponible.

Conclusión

Cuando conocí la noción de lenguajes algorítmicos nunca desafié la opinión, prevalente entonces, de que los problemas de diseño e implementación de lenguajes eran en su mayoría un asunto de compromiso: cda nueva ventaja para el usuario tenía que ser pagada por la implementación, ya sea a través de incremento de los problemas de traducción, o durante la ejecución, o durante ambos. Ahora bien, estamos bastante seguros de no estar viviendo en el cielo y no voy a negar la posibilidad de un conflicto entre conveniencia y eficiencia, pero ahora protesto cuando este conflicto se presenta como el resumen completo de la situación. Soy de la opinión que vale la pena investigar hasta que grado las necesidades del Hombre y de la Máquina van de la mano y ver que técnicas podemos divisar para el beneficio de todos nosotros. Confío que esta investigación producirá frutos y si esta charla ha hecho a algunos de ustedes compartir este ferviente deseo, ha logrado su objetivo.