CONSTRUCCIÓN DE UN DEPURADOR PORTABLE DE CÓDIGO. UNA VISIÓN GENERAL

José L. Arjona Fernández, José M. Prieto Pérez, R. Corchuelo Gil
Dpto. de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadítica
Universidad de Sevilla
Web: http://www.lsi.us.es

En esta serie de artículos presentaremos una de las herramientas más codiciadas por los programadores, el Depurador de Código. En este primer artículo se dará una visión global de que es un depurador y de las posibilidades que le ofrece al programador; se explicará el funcionamiento de los depuradores existentes y la característica más importante del depurador que pretendemos construir: *la portabilidad*.

¿Qué es un depurador portable de código?

Una de las inquietudes más importantes para los programadores ha sido la realización de programas libres de errores, es decir, desarrollar programas que funcionen correctamente atendiendo a unas especificaciones iniciales. Dado que es imposible a priori escribir programas sin errores, necesitamos de herramientas que sean capaces de facilitarnos la detección y eliminación de los errores, es decir, aparece la idea del depurador de código.

El depurador de código es una de las herramientas más apreciadas que acompañan al compilador y permitirá que el usuario del compilador pueda depurar simbólicamente sus programas, es decir, podrá detectar y corregir errores.

El depurador de código nos permitirá ver lo que ocurre dentro del programa cuando éste se ejecuta, es decir, nos permitirá seguir la evolución del estado del programa, ejecutarlo paso a paso, detenerlo en un determinado punto, visualizar el valor de las variables de una determinada rutina, podremos cambiar el valor de estas variables para ver como afecta al comportamiento del programa, realizar cálculos de expresiones, etcétera. Y lo que es más importante, mientras el programa depurado se ejecuta en segundo plano.

Por lo tanto la aparición de los depuradores, acaban con las antiguas técnicas de depuración, cómo pueden ser aquellas en las que el programador mostraba por pantalla el valor de ciertas variables interesantes ó críticas para la ejecución del programa.

Otro método muy extendido es el uso de trazas, que pretenden, normalmente, obtener de forma ordenada información acerca de la ejecución del programa, quedando registrada todas las entradas a funciones, los parámetros con las que se llamaron, valores de ciertas variables, etc ... Normalmente se suele trabajar con trazas multiniveles de forma que el programador puede seleccionar en que nivel de traza se quiere situar (Ej: Mostrar únicamente las llamadas a las funciones y los parámetros, ...).

```
int Suma(int a, int b) {        /* Suma el valor de 2 variables */
        int res;

        Traza_func_init("Suma");    /* Se indica que se entra en una función*/
```

```
        Traza_par("a",a);       /* Parámetro a */
        Traza_par("b",b);       /* Parámetro b */

        Res = a+b;

        Traza_ret("Res",Res); /* Muestra el valor de la var Res */
        Traza_func_end("Suma");     /* Se indica que se sale de una función*/

        return Res;
    }
```

Para el ejemplo anterior, y una vez que se ejecuta el programa, se podría generar la siguiente traza:

```
** Entrando en Función:  Suma
Parámetro a  = 2
        Parámetro b  = 3
        Valor de retorno Res  = 5
** Saliendo de Función:  Suma
```

Las técnicas anteriores adolecen de una característica que hacen que el depurador de código sea la herramienta perfecta para la detección de errores, esta característica es la flexibilidad. Es decir, el usuario no puede interactuar con el programa en ejecución; la detección de errores se convierte, por tanto, en un proceso estático y bastante rígido en la que el usuario no participa dinámicamente.

No obstante hay casos en los que además del depurador de código es necesario hacer uso de los métodos anteriores, aunque estos sean menos flexibles, por ejemplo para el uso en sistemas empotrados ó en la depuración de aplicaciones en las que la ejecución puede pasar por un gran número de estados (como pueden ser el desarrollo de un protocolo de telecomunicaciones, ...), en estos casos se suele dejar ejecutándose el programa durante meses mientras se genera la traza.

Errores.

Como se ha comentado anteriormente, la función del depurador es la de ayudar al programador a detectar los distintos errores que pueden aparecer en el programa.

Si el compilador se encarga de detectar los distintos errores léxicos, sintácticos y semánticos del código fuente, hay errores que no puede detectar, estamos hablando de los errores lógicos. Es decir, los errores producidos cuando el programador no codifica correctamente el algoritmo que resuelve un determinado problema. Estos fallos provocan salidas inesperadas e incorrectas del programa, aunque también pueden terminar con la pérdida de información ó bloqueo del ordenador.

Un ejemplo de este tipo de errores puede ser:

```
int i;
int j;
...
for(i=0;i<=10;j++) {
        printf("%d",i);
```

```
}
```

Con el ejemplo anterior, el compilador no daría ningún mensaje de error, sin embargo el programa nunca terminaría, ya que estamos ante un bucle infinito.

Posibilidades que ofrece el depurador.

A continuación vamos a ver las posibilidades que le ofrece el depurador al programador, es decir, ¿cómo y de qué forma ayuda el depurador en el proceso de detección de errores?

En primer lugar, y sin duda alguna, el servicio más importante que ofrece es la posibilidad de trazar el programa. El usuario podrá ejecutar paso a paso el programa, pudiendo preguntar en cada momento por el valor de las variables, parámetros, etc ... También es común en los depuradores, dar la posibilidad al programador de dejar de trazar las llamadas a las funciones que no contengan errores, es decir, que previamente haya depurado y comprobado que son correctas.

Veamos un ejemplo de traza:

```
10    i = 5;
11    j = 4;
12    k = Suma(i , j);
13    printf("%d" , k);
```

Para el código anterior un ejemplo de traza podría ser:

```
10:  i = 5;
(Suma.c:10)? watch i
    (int) 5
(Suma.c:10)? next
11:  j = 4;
(Suma.c:11)? watch 4+i
    (int) 9
(Suma.c:11)? next
12:  k = Suma(i , j);
(Suma.c:12)? step
13: printf("%d", k);
(Suma.c:13)? watch k
    (int) 9
...
```

Para la traza anterior se puede observar como el programador pregunta por el valor de la variable "*i*", además decide cuando se tiene que ejecutar la siguiente instrucción (*next*) o saltarse la depuración de una llamada a una función (*step*).

Por lo tanto, el programador podrá controlar la traza del programa, también tendrá la posibilidad de consultar el valor de las variables, usar estos valores en expresiones, y asignarle valores. Para poder hacer todo lo anterior el depurador necesitará:

- Tener información de todos los elementos que aparecen en el programa. Esta información vendrá descrita atendiendo a un formato y serán simples cadenas de caracteres denominadas stabs[1].

- Necesitaremos por tanto de un análisis léxico, sintáctico que nos permita reconocer los distintos stabs, así como almacenarlos de forma que nos proporcione un recorrido eficiente.

- También tendremos que ser capaces de evaluar expresiones, para ello tendremos que construir un evaluador de expresiones para el depurador de código.

Otra posibilidad que nos ofrecen los depuradores simbólicos, es la de los puntos de ruptura, cuyo cometido es parar al programa al llegar a cierto lugar. La utilidad más inmediata es la de llevar la traza de ejecución a un determinado lugar, al lugar donde se piensa que puede estar el error.

La funcionalidad de los puntos de ruptura se ve considerablemente aumentada con los puntos de ruptura condicionales. Lo que se pretende con un punto de ruptura condicional es que el programa se pare en un determinado lugar siempre y cuando se cumpla cierta condición. De esta forma nos sería fácil comprobar, por ejemplo, las precondiciones y postcondiciones de los bucles.

Funcionamiento de los depuradores actuales.

El funcionamiento de los depuradores actuales esta basado en uno de los modos de operación que tienen los procesadores, el modo *trap* que hace que el procesador lance una interrupción cada vez que se ejecuta una instrucción.

Además de lo anterior, y tal y como dijimos anteriormente, el depurador tiene que tener conocimiento de todos los elementos (variables, funciones, parámetros...) del programa fuente. Esta información es especificada nuevamente por los *stabs* y es generada por el compilador cuando se activa la opción de depuración. Las cadenas de stabs vienen definidas por directivas de ensamblador y son almacenadas en un segmento especial (Arquitecturas x86) a la que el depurador podrá acceder en todo momento.

Ya solo quedaría programar la rutina de servicio de la interrupción que genera el compilador automáticamente, para poder realizar las tareas típicas de un depurador de código, es decir: trazar, consultar el valor de las variables, cálculo de expresiones etc...

Característica del depurador simbólico de código que pretendemos construir: "portabilidad".

El depurador de código es un complemento de otra herramienta, el compilador. Un compilador genera a partir de un programa escrito en un lenguaje de programación definido, la traducción de éste a otro lenguaje, generalmente de más bajo nivel que el primero (respecto a dependencia de la máquina se refiere). Normalmente, y es lo más conocido por todos, esta

[1] Stab hace referencia a "symbol table", e identifica al formato general desarrollado por Peter Kessler en la Universidad de Berkeley (California, Estados Unidos), para representar la información de depuración.

traducción se realiza al lenguaje de la máquina en la que se ejecutará el programa. A esta traducción se le conoce como código objeto o ensamblador, y depende enormemente de la máquina para la que se genera código.

Debido a esta dependencia, el coste de adaptación de una plataforma hardware a otra es alto. Es decir, el esfuerzo de obtener versiones del compilador para diferentes máquinas o sistemas operativos llega a ser excesivo; además el código objeto generado no cuenta con una propiedad a la que se está tendiendo en la actualidad. Nos referimos a la portabilidad. Ésta es una característica que sería deseable que poseyera todo programa por la cual se puede transportar de una máquina a otra distinta sin ningún requisito previo (o por lo menos unos requisitos mínimos). Por este motivo, actualmente muchos compiladores generan código ANSI C como código objeto, en vez de generar código máquina. El estándar ANSI C es actualmente muy estable; los compiladores lo siguen al pie de la letra y a partir de él se puede llegar a generar un código máquina casi tan eficiente como el que un programador en ensamblador podría generar manualmente. Puesto que ANSI C es un estándar (muy popular a nivel de la comunidad de programadores) que soportan la amplia mayoría de compiladores de C de cualquier plataforma hardware, el código "objeto" generado es portable. Hemos ganado por tanto portabilidad a cambio de tener que compilar de nuevo dicho código objeto para obtener un programa ejecutable definitivo.

En la actualidad existen multitud de depuradores comerciales y académicos de gran calidad. Ahora bien, el principal “problema” de estos depuradores es que no permite depurar simbólicamente programas para aquellos compiladores que generan código C intermedio, ya que si usamos el depurador sobre estos programas lo que realmente estaremos depurando será el código C generado y no el código fuente original. Esto hace que muchos lenguajes como el SmallEiffel e Ingres 4GL carezcan de
depuradores simbólicos, siendo la única alternativa para encontrar los errores el incluir variable en determinados puntos, mostrar el valor de variables y parámetros, etc. pero en ningún caso depurarlo simbólicamente.

En definitiva, con la aparición de compiladores que generan código C en vez de código máquina, la tarea de depuración de un programa se puede llevar a cabo a dos niveles de abstracción como se puede observar en la figura siguiente:

- Depurar el código fuente original para el cual tenemos el compilador que genera código C.

- O bien depurar el código objeto en C que genera dicho compilador con las herramientas que existen en un buen entorno de desarrollo C y que todos los programadores suelen conocer (léase por ejemplo entornos de Borland o Microsoft).

Para tapar el hueco abierto debido a la falta de herramientas que se sitúen en el primer nivel de abstracción, nos planteamos el desarrollo de un depurador de código que trabaje a dicho nivel, y que bautizaremos con el nombre de Depurador Simbólico de Código Portable.

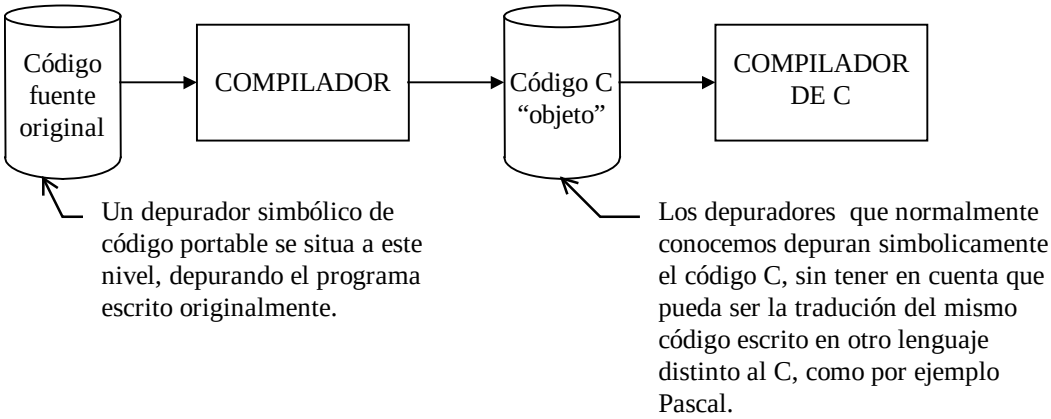

FIGURA 1. Niveles de abstracción en la depuración