Lenguajes de Programación I

Tipos de Datos Compuestos

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Conjuntos

La base de nuestra profesión

- Tipo de datos **Conjunto** (*set*) – propuesto inicialmente en Pascal.
  - Colección sin ordenar.
  - Tamaño arbitrario.
  - Elementos diferentes de un tipo común (**base** o **universo**).
- Operaciones típicas.
  - Unión.
  - Intersección.
  - Diferencia.
  - Vacuidad.

¿Cómo representarlos?

Sólo nos interesa saber si un elemento está presente o no...

- Arreglos.
- Tablas de hash.
- Arboles.
- Vector de Bits (la más frecuente).
  - Cardinalidad implícita en la longitud.
  - Presencia indicada con el bit correspondiente.
  - Operaciones muy eficientes usando operaciones lógicas del procesador.
    - Unión es el `or`.
    - Intersección es el `and`.
    - Diferencia es el `not` seguido de `and`.
  - No funcionan bien si el tipo base es grande.

Parecía una buena idea, pero resulta impráctica.

Tipos Recursivos

- Definición hace referencia a objetos del mismo u otro tipo.
  - Usualmente registros
  - Referencias adicionales a la información concreta.
- Naturales en la creación de estructuras *enlazadas* (listas, grafos, etc.).
- Modelo de referencia – se usa una referencia a otro registro.
- Modelo de valor – necesaria la noción de **apuntador**

Interpretar un valor como referencia a otro valor.

Apuntadores

Meta-valor

- **Apuntador** – nombre asociado a un valor que hace referencia a otro.
- Algunos lenguajes restringen el uso de apuntadores para que hagan referencia exclusivamente a objetos en el *heap*.
  - Función especial para *elaborar* un objeto en el *heap*.
  - En caso de éxito el valor retornado es un apuntador.
- Algunos lenguajes permiten crear apuntadores a objetos sin importar su modo de almacenamiento.
  - Función especial para *elaborar* un objeto en el *heap*.
  - Función especial para *tomar la dirección* de un objeto.
  - Siempre retorna un apuntador.

¡Un apuntador **no es equivalente** a una dirección en memoria!

Operaciones sobre Apuntadores

Siga la flecha...

- Operaciones frecuentes.
  - Reservar y liberar espacio en el *heap*.
  - Seguir el apuntador (*dereferencing*) hacia el objeto apuntado (*referent*).
  - Asignar el valor de un apuntador a otro.
  - Aritmética de apuntadores.
- Comportamiento varía
  - Lenguajes Funcionales vs. Lenguajes Imperativos.
  - Modelo de Referencia vs. Modelo de Valor.

Operaciones Comunes

- En Pascal sólo se puede apuntar al *heap*.

```
var p, r : ^integer;
new(p);
p^ := 42;
r := p;
writeln(r^);
```

- En C se puede apuntar a cualquier lado.

```
int bar = 69;
int foo (int a, float b) {
  int *pa; float *pb; int *px = &bar;
  pa = &a;
  pb = malloc(sizeof(float));
  *px = *pa + 42;
  *pb = b;
  return(*px);
}
```

Al árbol debemos...

- En Haskell

```
data Arb_E = Vacio | Nodo Int Arb_E Arb_E
```

- En Pascal

```
type arb_e_p = ^arb_e;
     arb_e = record
       izq, der : arb_e_p;
       val : integer;
     end;
```

- En C

```
struct arb_e {
  struct arb_e *izq, *der;
  int val;
};
```

Reservando espacio en el *heap*

Queremos que `ptr` apunte a un árbol

- En Pascal

```
ptr : arb_e_p;
new(ptr);
```

- En C

```
arb_e *ptr;
ptr = (arb_e *) malloc(sizeof(arb_e));
```

- En lenguajes orientados a objeto (Java)

```
Arb_E ptr;
ptr = new Arb_E ( ... )
```

Usando los apuntadores

- En Pascal existe un operador postfijo especializado.

```
ptr^.val := 42;
```

- En C existe un operador prefijo especializado...

```
(*ptr).val := 42;
```

- ...pero como casi siempre se apunta a registros

```
ptr->val := 42;
```

El mágico mundo de C

What sorcery is this?

```
int n;
int *a;
int b[10];

a = b;
n = a[3]
n = *(a + 3);
n = b[3];
n = *(b+3);
```

- Aritmética de apuntadores.
- ¡`[]` es un adorno para aritmética de apuntadores!
- Difícil determinar si dos *l-values* son aliases entre sí.

Referencia Colgante

I throw my arrows in the air, sometimes...

- El programa recupera el espacio de los objetos que salen de alcance.
- Para los objetos en el *heap* la recuperación es
  - Explícita (`dispose` en Pascal, `free` en C, `delete` en C++).
  - Implícita vía un *recolector de basura*.
- Un apuntador que hace referencia a un objeto recuperado se denomina **referencia colgante** (*dangling reference*).
  - Tratar de seguir el apuntador es un error a tiempo de ejecución.
  - ¿Pero cómo saber que se apunta a un espacio sin sentido?

Detectando Referencias Colgantes

- Asistido por el sistema de operación
  - Es un error apuntar fuera del `sbrk` o tope de pila – el sistema operativo genera señales SIGSEGV y SIGBUS.
  - Pero esto no puede detectar "flechas perdidas".
- **Lápidas** (*tombstones*)
  - Los apuntadores son indirectos.
  - Apuntan a una tumba, que apunta al objeto.
  - Si el objeto es liberado, la tumba se llena con un centinela.
  - Son costosas de implantar en espacio y tiempo.
- **Cerradura** (*locks and keys*).
  - Cada apuntador es un tupla (*direccion*, *llave*).
  - Cada objeto en el *heap* comienza con un *candado*.
  - Apuntador es válido, sólo si su llave coincide con candado del objeto.
  - Sólo sirven para objetos en el *heap*.

Recolección de Basura

- Recuperar el espacio cuando el objeto ya no es necesario.
- Original (LISP) y esencial en lenguajes funcionales – adoptada por lenguajes imperativos modernos.
- Difícil de implantar de manera efectiva en lenguajes imperativos, resultando más lenta que la recuperación explícita.
- ¿Cuándo deja de ser necesario un objeto?
  - Cuando nadie hace referencia a él.
  - Cuando no puede alcanzarse desde ningún objeto activo.

La pregunta es "¿este objeto ya no se usará más?" que desafortunadamente no es decidible.

Contar las Referencias

"Nadie me quiere"

- Cada objeto mantiene un contador de referencias – indica cuántos apuntadores hacen referencia a él.
- Comienza en uno al elaborar el objeto.
- Cuando se copian apuntadores:
  - Incrementa el contador del objeto apuntado por el *r-value*.
  - Decrementa el contador del objeto apuntado por el *l-value*.
- Cuando un apuntador sale de alcance:
  - Se decrementa el contador del objeto apuntado.
  - Si llega a cero, el objeto puede liberarse...
  - ...pero antes hay que decrementar los contadores de los objetos a los cuales apunta el objeto que estamos liberando!
- Complicación similar en el epílogo de las subrutinas.

Contar las Referencias

Un compromiso razonable

- ¿Cómo "detectar" apuntadores en una estructura? – necesario para el decremento recursivo.
  - Descriptores de tipos – para cada tipo y registro de activación.
  - Tabla con desplazamientos para las posiciones de apuntadores.
  - Valor especial *null* para apuntadores recién elaborados.
- Razonablemente predecible
  - Cuando ocurre, es en un cierre de alcance.
  - Duración proporcional a los objetos que salen de alcance
- No puede contra las referencias circulares.

Traza de Alcance – *Mark and Sweep*

"Téngase la bondad, suba los pies, toy' coleteando"

- Partir del Ambiente de Referencia Actual (variables vivas) y determinar cuáles bloques de memoria son accesibles.
- Consta de tres fases:
  1. Marcar **todos** los bloques del *heap* como inútiles – eso los hace susceptibles de recuperación.
  2. Partir del Ambiente de Referencia y explorar los apuntadores– cada bloque alcanzado *recursivamente* se marca como útil.
  3. Recorrer **todos** los bloques del *heap* – recuperar el espacio de los bloques inútiles que hayan quedado.
- Se dispara cuando no hay suficiente memoria para trabajar.
- Requiere que se suspendan los cambios al estado mutable.

Y esto tiene una serie de problemas

Traza de Alcance — *Mark and Sweep*

Problemas con el "proceso"

- Recorrer todos los bloques del *heap* – ¿dónde *comienza* cada bloque?
  - Tener bloques de tamaño fijo – ridículo. . .
  - Cada bloque necesita una marca de inicio, tamaño y marca de uso.
- Seguir recursivamente los apuntadores – ¿dónde están?
  - Cada bloque debe tener un apuntador a su descriptor de tipo.
- Seguir recursivamente los apuntadores. . .
  - Algoritmo recursivo que requiere una pila.
  - Profundidad de la pila proporcional a la cadena más larga de apuntadores en estructuras enlazadas.
  - ¿Y si no alcanza la memoria que queda para mantener esa pila? – si el recolector está corriendo, es porque no queda.
  - **Pointer Reversal** – al bajar recursivamente, cambiar la dirección del apuntador para "recordar" el camino de regreso.
    - Apuntador actual y previo, para implantar el proceso.
    - Campo adicional para indicar "apuntador de regreso".

Traza de Alcance — *Mark and Sweep*

Problemas con el "resultado"

- La recolección inocente produce fragmentación externa con rapidez – sabemos que es necesario compactar la memoria para combatirla.
- **Stop and Copy**
  - Usar solamente **la mitad** de la memoria del *heap*.
  - Cuando la mitad A se agota, corre el recolector.
  - Las páginas útiles de A son *copiadas* a la otra mitad B – se aprovecha de compactar las páginas para minimizar fragmentación.
  - La mitad A queda completamente vacía.
  - Se invierten los roles de A y B.

¡Sólo la **mitad** de la memoria!

Traza de Alcance — *Mark and Sweep*

The cost of collection is too damn high!

- Recorrer **todo** el *heap* es proporcional a su tamaño – más memoria disponible, más lento el recolector de basura.
- Estadísticamente, las reservas en el *heap* son de corta duración – esto es especialmente cierto en lenguajes funcionales.
- Dividir el *heap* en dos secciones – bloques "jóvenes" y "adultos"
  - El recolector trata de recuperar memoria en la sección "joven" – se supone que allí habrá más basura que recuperar (sin ofender).
  - Cuando un objeto "joven" sobrevive cierto número de recolecciones, se da por "maduro" y sus bloques pasan a la siguiente sección.
  - La promoción requiere convertir apuntadores al objeto "joven" de manera que apunten al objeto "adulto".
- ¿Cómo encontrar rápidamente apuntadores a "jóvenes" que deben convertirse en apuntadores a "adultos"?
  - En cada *asignación* el compilador genera código para verificar si el valor asignado es un apuntador a convertir.
  - Lista de conversión para futura referencia – *Write Barrier*

Traza de Alcance – *Mark and Sweep*

Una observación astuta permite ahorrar tiempo sacrificando pulcritud

- El problema de fondo es "encontrar" los apuntadores.
- No se apunta a cualquier dirección – reglas de alineación...
- Los apuntadores no están en cualquier parte – reglas de alineación...
- Si un patrón de bits *parece* un apuntador y está en una ubicación correctamente alineada, lo más *probable* es que sea un apuntador.
  - Revisar posiciones alineadas en la pila.
  - Si *parecen* una dirección en el *heap*, entonces es probable que ese bloque sea útil, y así se marca.
  - Se revisa ese bloque, con el mismo criterio.
- **Conservative Collection** – es rápida pero incompleta
  - *Nunca* va a marcar un bloque útil como basura.
  - *Puede* dejar pasar bloques de basura sin recuperar.
  - Es incapaz de compactar – no sabe si son o no son apuntadores.

Bibliografía

- [*Scott*]
  - Secciones 7.6 y 7.7
  - Ejercicios y Exploraciones
- Página de WikiPedia sobre Recolección de Basura
- Página en WikiPedia sobre el Recolector Boehm-Demers-Weiser – un recolector conservador para C/C++.