Lenguajes de Programación I

Subrutinas Genéricas - Excepciones

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Subrutinas Genéricas

Proveer la misma operación sobre tipos diferentes

- Pila de `X`, cola de `X`, diccionario de `X` – y no importan los detalles de `X`.
- **Polimorfismo Paramétrico** lo resuelve en Haskell – en otros lenguajes imperativos sigue habiendo problemas.
  - Declarar subrutinas con tipos incompletos pero consistentes.
  - Lentitud a tiempo de ejecución para verificar tipos (LISP)
  - Compilación compleja que además obliga a utilizar equivalencia estructural de tipos (ML).
- Mecanismo explícito de **Polimorfismo Genérico**.
  - Crear múltiples subrutinas a partir de una *plantilla* única.
  - Utiles para creación de *contenedores*.

Módulos genéricos con subrutinas genéricas.

Definición Genérica en C++

- Se declara una plantilla...

```
template<class item, int max_items = 100>
class queue {
  item items[max_items];
  ...
public:
  queue() { ... };
  void enqueue(item it) { ... };
  item dequeue() { ... }
}
```

- ...que se usa para crear específicos

```
queue<int,50> int_queue;
queue<process> ready_list;
```

Implantación puramente estática

Transformación antes y durante la compilación

- Emitir copias del código para cada instanciación – Ada, C++.
- Código de los métodos podría ser común a instancias del mismo tipo.
- Algunos lenguajes *garantizan* que todas las instancias comparten código a tiempo de ejecución – Java 5 lo hace apoyándose en `Object`.
- Similar a lo que haría un macro, pero con valores agregados:
  - Están integrados en el lenguaje.
  - Los tipos de los parámetros son verificados.
  - Los argumentos son evaluados una sola vez.
  - Las reglas de alcance aplican normalmente.

Restricciones en la parametrización

Genéricos específicos

- Interfaz o cabecera de declaración provee toda la información necesaria al programador usuario.
- Restricciones obligan a usar la plantilla de forma particular – sólo puede instanciarse con tipos para los que tenga sentido.
  - Restringiendo las operaciones disponibles – ¿basta que sean comparables? ¿basta que sean ordenables?
  - Con tipos que provean determinados métodos.
  - Con tipos que estén en determinada jerarquía de herencia.

Excepciones

- Condición inesperada o inusual, que surge durante la ejecución del programa y no puede ser manejada en el contexto local.
- Detectada implícitamente vs. generada (*raised*) explícitamente.
- El programador tiene tres opciones (todas mediocres):
  1. Inventar un valor que el llamador recibe en lugar de un valor válido – como el `NULL` en C y la cantidad de verificaciones...
  2. Retornar un valor de estado al llamador, que debe verificarlo – como `errno` en C y la cantidad de verificaciones...
  3. Pasar una clausura para una rutina que maneje errores – sólo en los lenguajes que soportan continuaciones.
- El manejo de excepciones resuelve el problema...
  - El caso normal se expresa de manera simple y directa.
  - El flujo de control se enruta a un *manejador de excepciones* sólo cuando es necesario.

Aproximación a las Excepciones

- Originalmente, se disponía de ejecución condicionada (PL/I)

```
ON condición
   instrucciones
```

- Los lenguajes modernos ofrecen bloques léxicos.
  - El bloque inicial de código para el caso normal.
  - Un bloque contigüo con el manejador de excepciones que *reemplaza* la ejecución del *resto* del bloque inicial en caso de error.
  - Las excepciones se propagan por la cadena dinámica.

Manejador de Excepciones

- En C++

```
try {
  // Caso normal
}
catch (...) {
  // Manejador
}
```

- En Perl

```
eval {
  # Caso normal
}
if ($@) {
  # Manejador
}
```

¿Qué se hace en el Manejador de Excepciones?

1. Si la excepción es recuperable, intenta recuperarse para continuar.
2. Si la excepción no puede ser recuperada en el bloque local, realiza la limpieza necesaria y luego propaga la excepción.
3. Si no puede recuperarse, al menos emite un mensaje de error razonable antes de terminar la ejecución.

¿Cómo se definen las Excepciones?

- Establecidas por el lenguaje durante su diseño e implantación – errores semánticos dinámicos que los programas pueden atrapar.
- Definidos por el programador
  - Usando un tipo predefinido en el lenguaje – `exception` en Ada.
  - Instanciando o especializando una clase – Java, Python.
  - Aproximación mixta (tipo base simple vs. librerías, Perl).
- Excepciones parametrizadas – incorporar detalles del problema.
- Se lanzan con instrucciones especiales.
  - `raise` – Ada, Modula-3, Python.
  - `throw` – Java, C++.
  - `die` o `croak` – Perl.

¿Cuál manejador atiende cada excepción?

Implantación obvia basada en anidamiento de llamadas

- Mantener una pila de listas de manejadores.
  - Al entrar a un bloque protegido, agregar manejador al principio.
  - Cuando ocurre una excepción, buscar en la lista del tope de la pila y utilizar los manejadores en orden.
  - Cada subrutina cuenta con un manejador implícito que ejecuta el epílogo y propaga la excepción.
- Costosa, incluso para el caso *normal*.
  - Agregar manejador a la lista, al entrar a la rutina.
  - Eliminar manejador de la lista, al salir de la rutina.
  - Búsqueda lineal en la lista por cada registro de activación.

¿Cuál manejador atiende cada excepción?

Implantación astuta basada en ordenar el código

- El código ejecutable se genera en bloques contiguos.
- Generar una tabla de manejadores a tiempo de compilación.
  - Cada entradas tiene dos direcciones
    - Dirección de inicio del bloque protegido.
    - Dirección de inicio del manejador correspondiente.
  - Se ordenan por el primer elemento.
  - Si ocurre una excepción
    - Búsqueda binaria de la dirección actual usando el *program counter*.
    - Si se propaga la excepción, se repite la búsqueda – después de todo, un manejador es un bloque de código...
  - Caso borde – manejadores implícitos para limpiar al salir deben usar la dirección de retorno como ubicación del error.
- El costo de ejecución aumenta sólo cuando ocurre una excepción – el caso normal no incurre en costo de ejecución.

Excepciones sin excepciones

Hay maneras de simularlas o implantarlas

- `goto` a etiquetas fuera de la subrutina actual (Pascal).
- `call-with-current-continuation` (Scheme)
  - Es una función `call/cc` que recibe una función `f`.
  - Llama a `f` pasando una clausura que captura el *program counter* y ambiente de referencia – esto es la **continuación**.
  - De ser necesario, `f` *podría* utilizar dicha clausura para reestablecer el ambiente de referencia.
- La pareja `setjmp` y `longjmp` (C).

```
if (!setjmp(buffer)) {
  /* Bloque que quizás usa longjmp */
} else {
  /* Manejador */
}
```

`setjmp` y `longjmp`

I don't usually go back in time, but when I do I save everything...

```
main()
{
  jmp_buf tardis;
  int i;
  i = setjmp(tardis);
  printf("i = %d\n", i * 2);
  if (i != 0) exit(0);
  longjmp(tardis, 21);
  printf("The cake is a lie!\n");
}
```

- `setjmp` salva el "estado" y retorna cero.
- `longjmp` restaura el "estado" y hace que `setjmp` retorne el argumento suministrado.

Co-rutinas

- **Co-rutina**
  - Se representa como una clausura.
  - Podemos saltar *dentro* haciendo una **transferencia** (*transfer*).
  - Las transferencias *conservan* el contador de programa, de modo que nuevas transferencias continúan donde habían quedado.
- Son ejecuciones
  - Que existen concurrentemente.
  - Ejecutan una a la vez.
- Se usan para implantar iteradores e hilos (*threads*).

Productor-Consumidor

El ejemplo canónico

```
var q := new Queue

coroutine produce
loop
    while q is not full
        /* crear items nuevos y ponerlos en q */
    transfer to consume

coroutine consume
loop
    while q is not empty
       /* tomar items de q y usarlos */
    transfer to produce
```

Implantación

Hace falta más de una pila

- Pila de ejecución.
  - Cada co-rutina tiene un espacio para su pila (estático o en el *heap*).
  - Se produce un error a tiempo de ejecución si se excede.
  - Si las co-rutinas pueden anidarse se usa una pila *cactus*.
- Operación de transferencia.
  - Conservar el contador de programa, la pila y los registros.
  - Al comenzar el `transfer` se empilan en el origen los registros a salvar y la dirección de retorno.
  - Se hace apuntar el stack pointer a la pila destino.
  - Se desempila la dirección de retorno y los registros salvados.

Bibliografía

- [*Scott*]
  - Secciones 8.4 a 8.6
  - Ejercicios y Exploraciones