Subrutinas y abstracción de control

Joaquín Rossi

Abstracción de control

Esconder una implementación compleja atrás de una interfaz simple.

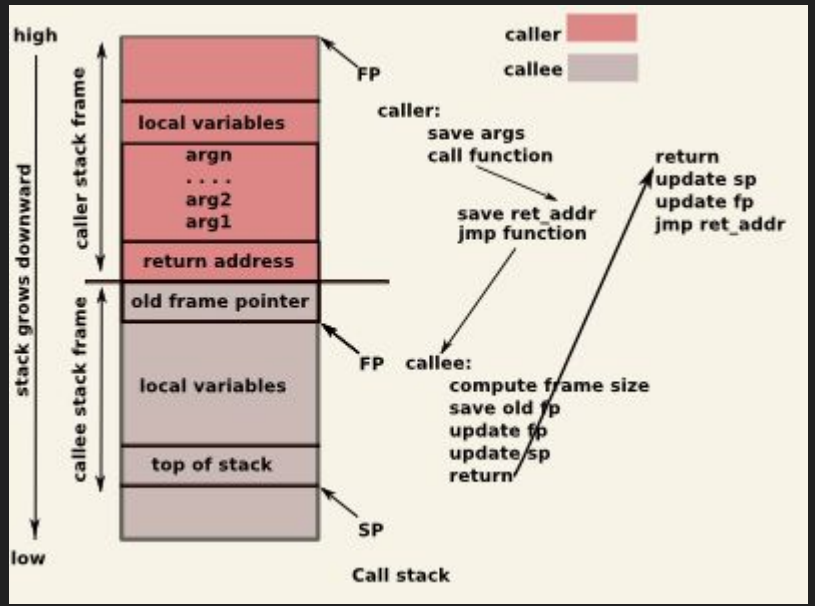
El mecanismo principal para la abstracción de control son las *subrutinas*: un código llamado (*callee*) realiza una acción para un llamante (*caller*), que espera hasta que termine.

- Usualmente esta operación es parametrizada, el caller pasa argumentos que influencia el comportamiento o le dan datos para trabajar. Estos argumentos o parámetros efectivos se mapean a los parámetros formales.
- Pueden retornar valores, se llaman *funciones* si lo hacen y *procedimientos* si no.
- Los lenguajes tipados estáticamente normalmente requieren una declaración para cada subrutina, con los tipos de sus parámetros y el de retorno.

Stack Layout

Cada vez que se llama a una subrutina se reserva un espacio en la cima del stack llamado *stack frame* o *activation record* para almacenar los argumentos, variables locales, la dirección de retorno, variables locales, temporales, valores previos de registros, etc.

Se tiene el *stack pointer* que apunta a la cima del stack, y el *frame pointer* que apunta al frame actual. Los elementos de tamaño constante se ubican al inicio y se acceden con un desplazamiento desde el *fp*, y si no se usa un dope vector que guarda su posición y se accede a través de él.



Subrutinas anidadas

Para las subrutinas anidadas con scoping estático se mantiene un *static chain* con una referencia (*static link*) al frame de la subrutina superior.

En el caso de las clausuras se le pasa una referencia a una copia del valor del frame del padre cuando la clausura fue creada (el *dynamic link*).

```
function add(x: int, y: int): int
     function addX(y: int): int
          return x + y;
     end
     return addX(y);
end
```

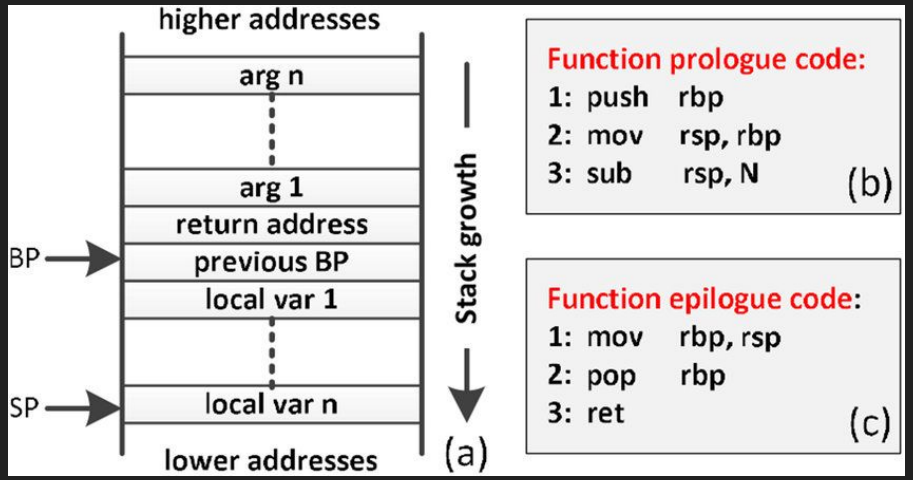
```
function addX(
     paramsAdd: ^{x: int, y: int},
     y: int
): int
     return paramsAdd^.x + y;
end

function add(x: int, y: int): int
     return addX(@params, y);
end
```

Calling sequences

Para mantener el call stack se tiene que hacer cierto trabajo antes y después de ejecutar el código de una subrutina (el *prólogo* y el *epílogo*).

- Prólogo
  - Pasar argumentos
  - Setear dirección de retorno
  - Saltar al código llamado (cambiar program counter)
  - Reservar espacio (cambiar stack pointer)
  - Guardar registros modificables por el callee
  - Cambiar el frame pointer al frame nuevo
  - Ejecutar código de inicialización de objetos del frame nuevo
- Epílogo
  - Retornar valores
  - Ejecutar código de finalización de objetos del frame
  - Liberar espacio reservado
  - Volver al código llamante



Varias de estas tareas las puede hacer uno o el otro, pero si las hace el callee se ahorra espacio en general ya que solo se tiene que hacer una vez por función, y cada función en general se llama más de una vez.

Quién hace cada una depende de la *calling convention*.

La división más complicada de esta tarea reside en guardar los registros, ya que sería ideal solo tener que guardar los que se están usando en el caller, pero debido a la complicación separada no se puede y se selecciona qué registros debe guardar cada uno.

In-line expansion

Una alternativa a la llamada en stack, donde el código se reemplaza donde es llamado.

Esto evita varios overheads y permite más optimizaciones, pero:

- No permiten la compilación separada
- Pueden llevar a que el programa ocupe más espacio
- No se pueden usar en general para las subrutinas recursivas.

Se parecen a las macros, pero no tienen los problemas semánticos de estas, ya que son solo un detalle de implementación (misma semántica que llamadas por stack).

Pasaje de parámetros

Las maneras en las que se pasan los argumentos.

- **Call by value**: se pasa una copia (normalmente shallow)
- **Call by reference**: se pasa un puntero (usualmente se debe pasar un l-value)
  - En lenguajes que no lo tienen (como C) se puede simular pasando un puntero manualmente.
  - Read-only: para obtener la eficiencia de las referencias pero sin dejar que se modifique se puede usar un puntero const.
- **Call by value/result**: copia el valor a al parámetro al llamar y copia el resultado al argumento al volver.
- **Call by sharing**: usado en lenguajes con modelo de referencias para variables, se pasa la referencia y se permite modificar el objeto, pero no se puede hacer que apunte a otro.

Referencias en C++

L-value references

Similares a los punteros, pero sin la sintaxis especial y no se las puede hacer apuntar a otro objeto.

```
int i;
int &j = i;
j = 2;
cout << i; // imprime 2
```

R-value references

Permiten pasar un r-value (normalmente una expresión) por referencia, para poder *mover* los valores desde objeto sin la necesidad de copiar.

```
std::vector<int> xs = {1, 2, 3};
std::vector<int> ys = xs; // copy constructor
std::vector<int> zs = std::move(xs); // move constructor

// std::move es esencialmente T& -> T&&
```

Se "roba" los valores apuntados por el objeto desde el que se mueve sin copiar, ya que se sabe que no los va a usar más.

Clausuras como parámetros

Se implementan como un puntero a una función + un puntero al ambiente (el dynamic link) que se pasa como uno de los argumentos.

Permiten nuevos modos de pasaje de parámetros

- **Call by name**: se pasa un *thunk* (clausura que computa el valor del parámetro) y solo se utiliza cuando se necesita.
- **Call by need**: similar a call by name pero solo se calcula *la primera vez* que se necesita el valor.

NAME

qsort, qsort_r — sort a table of data

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
       int (*compar)(const void *, const void *));

[CX] ⌦ void qsort_r(void *base, size_t nel, size_t width,
       int (*compar)(const void *, const void *, void *), void *arg); ⌫
```

Diferencias sintácticas

Parámetros opcionales:

Permiten omitir ciertos valores, dándoles un valor por defecto.

```
def greet(name="friend"):
    return "Hello, " + name

greet() # "Hello, friend"
greet("John") # "Hello, John"
```

Parámetros nombrados

En lugar de pasar en un orden se pueden poner de cualquier manera, identificandonos con un nombre, permitiendo evitar errores de orden que el sistema de tipos no atrapa.

```
def area_cylinder(radius, height):
    return math.pi * radius * radius * height

area_cylinder(height=1.0, radius=2.0)
```

Se suelen usar en conjunción con los parámetros opcionales.

Número variable de argumentos

El ejemplo por excelencia es la función *printf* de C. Se implementan pasandolos como un array de tamaño variable.

Deben ponerse al final de los demás parámetros.

```java
void print_lines(String... lines) {
    for (String line : lines) {
        System.out.println(line);
    }
}

print_lines("Line 1", "Line 2", "Line 3");
```

```
template<typename T,
   typename... Args>

void print(T first,
          Args... args){
    print(first);
    print(args...);
}
```

Retorno de una función

En lenguajes que no distinguen expresiones y sentencias, el valor de la función es el valor de la expresión de su cuerpo.

En lenguajes imperativos se suele tener una sentencia `return <expression>` o se puede asignar a una variable especial con el nombre de la función. Esta sentencia además devuelve el control al caller.

Algunos lenguajes permiten darle un nombre especial al valor retornado, que se usa como variable local y es retornada automáticamente al final:

```
func A_max(A []int) (rtn int) {
    rtn = A[0]
    for i := 1; i < len(A); i++ {
        if A[i] > rtn { rtn = A[i] }
    }
}
```

¡

En otros se puede retornar más de un valor:

```
def foo():
    return 2, 3

i, j = foo()
```

Excepciones

Mecanismo que permite alterar el flujo del programa en casos de situaciones inesperadas.

En caso de que una sea causada, se permite al código caller "atrapar" la excepción y ejecutar algo distinto.

En caso de que no sea atrapada en la subrutina donde se tira, avanza por el call stack hasta encontrar algún punto donde sí es manejada, alterando el flujo del programa al no retornar un valor desde la subrutina de origen.

En muchos lenguajes se pueden manejar las excepciones como valores, de tipos que el usuario puede definir, y puede atrapar dependiendo del tipo y manejarlas de distintas maneras.

```
try {
    ...
} catch (IOException e) {
    ...
} catch (Exception e) {
    ...
}
```

Cleanup

Al regresar hacia frames anteriores el mecanismo debe hacer "unwind" del stack, liberando los frames. Ya que algunos de los objetos liberados puede necesitar ejecutar una lógica especial aunque se tire una excepción (como liberar recursos).

```
try {
    ...
} catch (Exception e) {
    ...
} finally {
    ...
}
```

Esto puede ser evitado si el lenguaje cuenta con destructores (que se deben ejecutar al hacer unwind) o con un bloque especial de manejo de recursos.

```
try (FileReader fr = new FileReader(path);
     BufferedReader br = new BufferedReader(fr)) {
    return br.readLine();
}
```

```
with open("/etc/passwd", "r") as f:
    print f.readlines()
```

Manejo de excepciones como expresiones

En lenguajes orientados a expresiones se hace este control vía expresiones en lugar de sentencias, que deben retornar un tipo compatible con el de la expresión "principal":

```
let foo =
    try a / b
    with Division_by_zero -> max_int;;
```

A la expresión de *throw* se le puede asignar el tipo especial Bottom (también llamado Never) que no contiene valores y es subtipo de todos los tipos (opuesto a Top).

```
int one_over_x = x != 0 ? 1 / x : throw
error();
// el tipo de one_over_x es la unión de 1/x
(int) y never, que resulta en int.
```

Implementación de excepciones con setjmp/longjmp

```
#include <stdio.h>

#include <setjmp.h>


#define TRY do { jmp_buf ex_buf__; if ( !setjmp(ex_buf__)){

#define CATCH } else {

#define ETRY } } while(0)

#define THROW longjmp(ex_buf__, 1)
```

```
int main(int argc, char** argv) {
    TRY {
        printf("In Try Statement\n");
        THROW;
        printf("I do not appear\n");
    } CATCH {
        printf("Got Exception!\n");
    } ETRY;
    return 0;
}
```

https://web.archive.org/web/20240822013656/https://groups.di.unipi.it/~nids/docs/longjump_try_trow_catch.html

Corrutinas

Clausuras a las cuales se entra varias veces en distintas puntos, que pueden transferirse el control entre sí y se ejecutan concurrentemente.

Cada una requiere su propio stack ya que el pasaje de control hace que pueda pasar que no se respete el orden LIFO entre corrutinas.

Al pasar de una corrutina a la otra se debe cambiar el stack y preservar los valores necesarios.

Ya que pueden ser declaradas a cualquier nivel de anidación, dos subrutinas pueden compartir partes anteriores de sus stacks, que deben compartir, llevando al *stack cactus*.

```
var q := new queue

coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume
coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
call produce
```

Fin