

Metody Realizacji Języków Programowania

Realizacja funkcji, procedur i metod

Marcin Benke

MIM UW

6 grudnia 2010

Realizacja funkcji i procedur

- Instrukcja wywołania funkcji nakazuje rozpoczęcie wykonania funkcji, a po jej zakończeniu powrót do instrukcji za wywołaniem.
- Historię obliczeń w programie można przedstawić w postaci tzw. drzewa aktywacji, którego węzły reprezentują wcielenia (wykonania) funkcji.

Drzewo aktywacji

- Korzeń tego drzewa to wykonanie programu głównego a węzeł F ma synów $G_1 \dots G_n$ jeśli wcielenie funkcji F wywołało G_1 , później G_2 itd.
- Podczas wykonania programu odwiedzamy węzły porządku prefiksowym, od lewej do prawej.
- Na ścieżce od korzenia do aktualnego węzła są aktywne wcielenia funkcji, na lewo już zakończone a na prawo jeszcze się nie rozpoczęte.
- Jeśli istnieje ścieżka, na której występuje wiele wcieleń tej samej funkcji, mówimy że funkcja ta jest rekurencyjna.

Przekazywanie sterowania

- Są dwa sposoby realizacji przekazywania sterowania między wołającym a wołanym:
 - ▶ podprogram otwarty (ang. inline, macro):
kod funkcji/procedury wołanej wstawiamy w miejscu wywołania
 - ▶ podprogram zamknięty:
przekazanie sterowania następuje za pomocą instrukcji skoku ze śladem
- Podprogram otwarty może dać bardziej efektywny kod — eliminuje instrukcje skoku.
- W większości przypadków wydłużenie kodu wynikowego, mniej uniwersalna metoda — nie można jej zastosować do funkcji rekurencyjnych.
- W dalszej części wykładu zajmiemy się podprogramami zamkniętymi.

Rekord aktywacji

- Z każdym wcieleniem funkcji wiążemy pewne informacje. Obszar pamięci, w którym są zapisywane, nazywamy *rekordem aktywacji* albo *ramką* (ang. *frame*).
- W większości języków potrzebne są tylko rekordy dla aktywnych wcieleń na aktualnej ścieżce w drzewie aktywacji.
- Gdyby nie rekurencja, dla każdej funkcji moglibyśmy z góry zarezerwować obszar pamięci na jeden rekord aktywacji, gdyż wiemy, że tylko tyle rekordów będzie potrzebnych (patrz wczesny Fortran).
- W językach z rekurencją rekordy aktywacji alokujemy przy wywołaniu funkcji a zwalniamy, gdy funkcja się skończy. Rekordy aktywacji przechowujemy więc na stosie.

Zawartość rekordu aktywacji

Informacje pamiętane w rekordzie aktywacji zależą m. in. od języka. Mogą tam być:

- parametry
- zmienne lokalne i zmienne tymczasowe
- ślad powrotu
- kopia rejestrów (wszystkich, niektórych lub żadnego)
- łącze dynamiczne (DL, ang. dynamic link) – wskaźnik na poprzedni rekord aktywacji; ciąg rekordów połączonych wskaźnikami DL tworzy tzw. łańcuch dynamiczny.
- łącze statyczne (SL, ang. static link)
- miejsce na wynik

Postać rekordu aktywacji nie jest sztywno określona — projektuje ją autor implementacji języka.

Adresowanie rekordu aktywacji

- Adres ramki jest zwykle przechowywany w wybranym rejestrze (FP = frame pointer, BP = base pointer).
- Pola rekordu aktywacji są adresowane przez określenie ich przesunięcia względem adresu w FP.
- Każde wcielenie funkcji, niezależnie od położenia rekordu aktywacji, w ten sam sposób może korzystać z jego zawartości, a więc wszystkie wcielenia mają wspólny kod.
- Adresem rekordu aktywacji nie musi być adres jego początku. Jeśli uznamy, że tak będzie wygodniej, możemy przyjąć, że adresem rekordu aktywacji będzie adres jednego z pól w środku tego rekordu.

Adresowanie rekordu aktywacji

- Jeśli w języku występują funkcje ze zmienną liczbą argumentów (jak np. w C), adresowanie rekordu aktywacji względem jego środka może być najwygodniejszym rozwiązaniem.
- Gdyby pola rekordu aktywacji były adresowane względem jego początku, przesunięcia pól zależałyby od liczby parametrów, a więc nie byłyby znane podczas kompilacji.
- Rozwiązaniem tego problemu jest adresowanie rekordu aktywacji względem miejsca pomiędzy parametrami a zmiennymi lokalnymi funkcji.
- Parametry wkłada się do rekordu aktywacji w kolejności od ostatniego do pierwszego, dzięki czemu przesunięcie miejsca, w którym znajduje się K-ty parametr, nie zależy od liczby parametrów, tylko od stałej K.
- Wynik funkcji często w rejestrach zamiast na stosie.

Pomijanie wskaźnika ramki

- Aktualny rekord aktywacji zawsze znajduje się na wierzchołku stosu rekordów aktywacji; można do jego adresowania użyć wskaźnika stosu.
- Zalety: oszczędzamy jeden rejestr i kilka instrukcji na każde wywołanie.
- Wady: wierzchołek stosu przesuwa się podczas obliczeń, powodując zmiany przesunięć pól rekordu aktywacji; podatne na błędy w generowaniu kodu.
- Rozwiązanie stosowane w JVM, także w GCC z opcją `-fomit-frame-pointer` (zatem przeważnie także z opcją `-O`)

Protokół wywołania (i powrotu z) funkcji

- Protokół wywołania opisuje czynności, które ma wykonać wołający (zarówno przed przekazaniem sterowania do wołanego, jak i po powrocie) oraz to, co wołany (czyli każda funkcja) ma robić na początku i na końcu.
- Podstawowym zadaniem jest zbudowanie rekordu aktywacji oraz usunięcie go.
- Niektóre czynności musi wykonywać wołający (np. liczenie parametrów), inne wołany (np. zarezerwowanie miejsca na zmienne lokalne), a jeszcze inne może wykonać zarówno wołany jak i wołający.

Protokół wywołania (i powrotu z) funkcji

Zaprojektujemy protokół wywołania i powrotu z funkcji. Założymy przy tym następującą postać rekordu aktywacji:

- miejsce na wynik
- parametry
- ślad
- DL
- zmienne

Protokół wywołania (i powrotu z) funkcji

Wskaźnikiem rekordu aktywacji będzie BP zawierający adres pola DL.wołający

```
PUSH 0 ; miejsce na wynik
<parametry na stos>
CALL adres_wołanego
ADD n, SP ; n - łączny rozmiar parametrów
; wynik zostaje na stosie
```

wołany

```
PUSH BP ; DL na stos
MOV SP, BP ; aktualizacja wskaźnika ramki
SUB k, SP ; k - łączny rozmiar zmiennych
... ; tłumaczenie treści funkcji
MOV BP, SP ; przywracamy wskaźnik stosu
POP BP ; przywracamy wskaźnik ramki
RET ; powrót do wołającego
```

Przykład

```
int sumto(int n)
{
    int i, sum;
    i = 0;
    sum = 0;
    while (i<n) {
        i = i+1;
        sum = sum+i;
    }
    return sum;
}
```

Z ramką stosu

```
sumto:  pushl    %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $0, -8(%ebp) ; i
        movl    $0, -4(%ebp) ; sum
        jmp     .L2

.L3:
        addl    $1, -8(%ebp)
        movl    -8(%ebp), %eax
        addl    %eax, -4(%ebp)

.L2:
        movl    8(%ebp), %eax ; n
        cmpl   %eax, -8(%ebp)
        jl     .L3
        movl    -4(%ebp), %eax
        leave   ; = movl ebp, esp; popl ebp
        ret
```

Bez ramki stosu

sumto:

```
movl    4(%esp), %ecx ; ecx = n
xorl    %eax, %eax    ; eax = 0
testl   %ecx, %ecx    ; ecx <= 0?
jle     .L4           ; skok do L4
xorl    %edx, %edx    ; edx = 0
```

.L5:

```
addl    $1, %edx      ; edx += 1
addl    %edx, %eax    ; eax += eax
cmpl    %edx, %ecx    ; edx != n?
jne     .L5           ; skok do L5
```

.L4:

```
ret     ; powrót
```

Interludium: maszyna stosowa

W dalszym ciągu wykładu będziemy się posługiwać prostą maszyną stosową, z operacjami:

- *stała* — połóż stałą na stosie
- **LOAD** — połóż na stosie wartość spod adresu zdjętego ze stosu
- **STORE** — zdejmuj ze stosu adres, następnie zdejmuj ze stosu wartość i wkładaj ją pod ten adres
- **ADD, SUB, ...** — zdejmuj ze stosu argumenty i zastępuj wynikiem operacji
- **GOTO** — skacze pod adres na stosie
- **CALL** — j.w, zostawiając na stosie ślad powrotu
- **SP** — kładzie na stosie adres wskaźnika stosu
- **FP** — kładzie na stosie adres wskaźnika ramki

Mechanizm przekazywania parametrów

Dwa najczęściej spotykane tryby przekazywania parametrów to:

- przekazywanie parametru przez wartość:
 - ▶ wołający umieszcza w rekordzie aktywacji wartość argumentu;
 - ▶ wołany może odczytać otrzymaną wartość oraz, jeśli język programowania na to pozwala, zmieniać ją traktując parametr tak samo, jak zmienną lokalną;
 - ▶ ewentualne zmiany nie są widziane przez wołającego.
- przekazywanie parametru przez zmienną (referencję)
 - ▶ wołający umieszcza w rekordzie aktywacji adres zmiennej;
 - ▶ wołany może odczytać wartość argumentu sięgając pod ten adres,
 - ▶ może też pod ten adres coś wpisać, zmieniając tym samym wartość zmiennej będącej argumentem.

Przykład 1

W programie, którego fragment wygląda następująco:

```
procedure p;  
var a,b : integer;  
begin  
  a := q(b,b)  
end;
```

```
function q(x : integer; var y : integer) : integer;  
var z : integer;  
begin  
  q := x + y;  
  y := 7  
end;
```

rekord aktywacji procedury p wyglądałby tak (w nawiasach podano przesunięcia poszczególnych pól względem pola DL wskazywanego przez rejestr FP):

(2) miejsce na wynik
(1) ślad
(0) DL
(-1) a
(-2) b

a rekord aktywacji funkcji q tak:

(4) miejsce na wynik
(3) x
(2) y
(1) ślad
(0) DL
(-1) z

kod wynikowy dla procedury p miałby postać:

```
FP LOAD                ; DL do ramki
SP LOAD FP STORE      ; nowy adres ramki do FP
0 0                   ; miejsce na dwie zmienne
0                     ; miejsce na wynik
FP LOAD -2 ADD LOAD   ; wartość b
FP LOAD -2 ADD        ; adres zmiennej b
q CALL                ; skok ze śladem do q
DROP DROP            ; usuwamy parametry
FP LOAD -1 ADD STORE  ; przypisujemy wynik na a
DROP DROP            ; usuwamy zmienne
FP STORE              ; wracamy do poprzedniej ramki
GOTO                  ; i do miejsca wywołania
```

a kod funkcji q:

```
FP LOAD                                ; DL do ramki
SP LOAD FP STORE                        ; nowy adres ramki do FP
0                                        ; miejsce na jedną zmienną
FP LOAD 3 ADD LOAD                      ; wartość x na stos
FP LOAD 2 ADD LOAD LOAD                 ; wartość y na stos
ADD                                      ; dodajemy x i y
FP LOAD 4 ADD                            ; adres miejsca na wynik
STORE                                    ; zapamiętujemy wynik
7                                        ; stała na stos
FP LOAD 2 ADD LOAD                      ; adres zmiennej
                                        ; przekazanej jako y
STORE                                    ; przypisanie na y
DROP                                     ; usuwamy zmienną
FP STORE                                ; wracamy do poprzedniej ramki
GOTO                                     ; i do miejsca wywołania
```

Środowisko w językach ze strukturą blokową

- Wiele języków programowania (n.p. Pascal) pozwala na zagnieżdżanie funkcji i procedur. Języki te nazywamy językami ze *strukturą blokową*.
- Kod funkcji ma dostęp nie tylko do jej danych lokalnych, ale także do danych funkcji, w której jest zagnieżdżona itd. aż do poziomu globalnego.
- Działanie funkcji jest określone nie tylko przez jej kod oraz parametry, lecz także przez środowisko, w którym ma się wykonać.

Wiązanie statyczne i dynamiczne

- Postać środowiska jest w Pascalu wyznaczona statycznie — z kodu programu wynika, do której funkcji należy rekord aktywacji, w którym mamy szukać zmiennej nielokalnej.
- Mówimy, że w Pascalu obowiązuje statyczne wiązanie zmiennych.
- Istnieją również języki (n.p. Lisp), w których obowiązuje wiązanie dynamiczne — w przypadku odwołania do danej nielokalnej, szukamy jej w rekordzie aktywacji wołającego itd. w górę po łańcuchu dynamicznym.

Łącze statyczne

- By korzystać z danych nielokalnych, działająca funkcja musi mieć dostęp do swojego środowiska.
- Moglibyśmy przekazać jej wszystkie dane znajdujące się w jej środowisku jako dodatkowe parametry. Rozwiązanie takie stosuje się często w językach funkcyjnych.
- W językach imperatywnych najczęściej stosowanym rozwiązaniem jest powiązanie w listę ciągu ramek, które są na ścieżce w hierarchii zagnieżdżenia.
- Każda ramka zawiera łącze statyczne (SL, static link) – wskaźnik do jednego z rekordów aktywacji funkcji otaczającej daną.
- Rekord ten nazywamy poprzednikiem statycznym, a ciąg rekordów połączonych SL to łańcuch statyczny.

Wyliczanie SL

- SL musi być liczony przez wołającego, bo do jego określenia trzeba znać zarówno funkcję wołaną jak i wołającą.
- Środowisko dla funkcji wołanej zależy od środowiska wołającej – jeśli obie widzą zmienną x , jej wartość ma być dla nich równa.
- Jeśli funkcja F znajdująca się na poziomie zagnieżdżenia F_p wywołuje G z poziomu zagnieżdżenia G_p , w pole SL wpisze adres rekordu, który odnajdzie przechodząc po własnym łańcuchu statycznym o $F_p - G_p + 1$ kroków w górę.
- Jeśli np. G jest funkcją lokalną F (czyli $G_p = F_p + 1$), funkcja F w pole SL dla G wpisze adres swojego rekordu; jeśli F i G są na tym samym poziomie zagnieżdżenia, w polu SL dla G będzie to, co w SL dla F itd.

Dostęp do danych nielokalnych

- Dane lokalne funkcji są w jej rekordzie aktywacji a dane globalne w ustalonym miejscu w pamięci — można do nich sięgać za pomocą adresów bezwzględnych.
- Z danych, które nie są ani lokalne ani globalne, korzystamy przy pomocy łańcucha statycznego. W funkcji F o poziomie F_p sięgamy do zmiennej z funkcji G o poziomie G_p (oczywiście $F_p \geq G_p$), przechodząc po łańcuchu statycznym $F_p - G_p$ kroków w górę. Tam pod przesunięciem znanym podczas kompilacji jest nasza zmienna. Adres zmiennej jest wyznaczony przez poziom zagnieżdżenia i przesunięcie w rekordzie.
- Adresy rekordów z łańcucha statycznego można też wpisać do tablicy (tzw. display). Dzięki temu unikniemy chodzenia po łańcuchu statycznym, ale za to trzeba będzie stale aktualizować tablicę.

Przykład 2

- Przyjmijmy, że w rekordzie aktywacji SL będzie się znajdował pomiędzy parametrami a śladem powrotu.
- Zakładamy też, że protokół wywołania i powrotu z funkcji jest prawie taki sam, jak w poprzednim przykładzie.
- Jediną różnicą będzie dodanie po stronie wołającego obliczenia SL przed wywołaniem i usunięcia go ze stosu po powrocie sterowania.

Przykład 2

W programie, którego fragment wygląda następująco:

```
procedure p;  
var a : integer;  
  procedure q;  
  var b : integer;  
    procedure r;  
    var c : integer;  
      procedure s; begin ... end {s};  
    begin  
      a:=b+c;  
      s;  
      q  
    end {r};  
  begin ... end {q};  
begin ... end {p};
```

Przykład 2 — rekord aktywacji p

rekord aktywacji procedury p wyglądałby tak (w nawiasach podano przesunięcia poszczególnych pól względem pola DL wskazywanego przez rejestr FP):

```
( 3 )  miejsce na wynik  
( 2 )  SL  
( 1 )  ślad  
( 0 )  DL  
( -1 ) a
```

Przykład 2 — rekordy aktywacji q,r

procedury q tak:

```
( 3 )  miejsce na wynik  
( 2 )  SL  
( 1 )  ślad  
( 0 )  DL  
( -1 ) b
```

a procedury r tak:

```
( 3 )  miejsce na wynik  
( 2 )  SL  
( 1 )  ślad  
( 0 )  DL  
( -1 ) c
```

Przykład 2 — kod procedury r

kod wynikowy dla procedury r miałby postać:

```
r:
FP LOAD                ; DL do rekordu aktywacji
SP LOAD FP STORE      ; nowy adres ramki do FP
0                      ; miejsce na zmienną c
FP LOAD 2 ADD LOAD    ; SL na stos
-1 ADD LOAD           ; wartość b na stos
FP LOAD -1 ADD LOAD   ; wartość c na stos
ADD                   ; b+c na stos
FP LOAD 2 ADD LOAD    ; SL na STOS
2 ADD LOAD            ; na stos adres ramki p
-1 ADD                ; na stos adres zmiennej a
STORE                 ; przypisanie
```

Przykład 2 — kod procedury r c.d.

```
0           ; miejsce na wynik
FP LOAD    ; SL dla s na stos
s CALL     ; skok ze śladem do s
DROP       ; usuwamy SL
DROP       ; usuwamy miejsce na wynik
0           ; miejsce na wynik
FP LOAD 2  ; adres ramki q
ADD LOAD   ; SL dla wywołania q
2 ADD LOAD ; skok ze śladem do q
q CALL     ; usuwamy SL
DROP       ; usuwamy miejsce na wynik
DROP       ; usuwamy zmienną c
FP STORE   ; wracamy do poprzedniej ramki
GOTO       ; i do miejsca wywołania
```


Realizacja konstrukcji języków obiektowych

- W obiektowych językach programowania każdy obiekt posiada pewną wiedzę, przechowywaną na zmiennych instancyjnych (zmiennych obiektowych), ma również określone umiejętności reprezentowane przez metody.
- To, jakie zmienne instancyjne i jakie metody posiada obiekt danej klasy, wynika z definicji tej klasy oraz z definicji klas, z których dziedziczy bezpośrednio lub pośrednio.
- W dalszej części wykładu omówimy realizację mechanizmów obiektowości dla języka programowania, w którym każda klasa może dziedziczyć z co najwyżej jednej klasy (języka z pojedynczym dziedziczeniem).

Zmienne instancyjne

- Każdy obiekt posiada zmienne zdefiniowane w jego klasie, a także zmienne odziedziczone z nadklas.
- Reprezentacja obiektów jest analogiczna do rekordów — w obszarze pamięci zajęтым przez obiekt znajdują się wartości jego zmiennych instancyjnych.
- Kolejność tych zmiennych ma być zgodna z hierarchią dziedziczenia — zmienne zdefiniowane w klasie obiektu muszą się znaleźć na końcu, przed nimi są zmienne z klasy dziedziczonej itd. w górę hierarchii dziedziczenia.

Np. w programie zawierającym definicje klas:

```
class A {
  var w:integer;
  procedure piszA { write(w) }
}
class B extends A {
  var x,y:integer;
  procedure piszB { write(w,x,y) }
}
class C extends B {
  var z:integer;
  procedure piszC { write(w,x,y,z) }
}
```

metody `pisz...` wypisują wartości wszystkich zmiennych obiektu danej klasy.

- W programie tym, obiekty klasy A będą zawierały jedną zmienną:
w
- obiekty klasy B trzy zmienne uporządkowane w kolejności:
w x y
- a obiekty klasy C cztery zmienne w kolejności:
w x y z
- Przyjęcie takiej kolejności jest konieczne, by umożliwić metodom danej klasy prawidłowe działanie zarówno dla obiektów tej klasy, jak i jej podklas.
- Musimy zagwarantować, by w obiekcie dziedziczącym zmienną znajdowała się ona w tym samym miejscu, co w obiektach klasy dziedziczonej.
- W naszym przypadku, zarówno w obiektach klasy B jak i klasy C, zmienne x i y są odpowiednio na 2 i 3 pozycji.
- Metoda piszB wie pod jakim przesunięciem te zmienne się znajdują niezależnie od tego, czy zostanie wywołana dla obiektu klasy B czy C.

Metody wirtualne

Wywołanie metody w językach obiektowych różni się od wywołania funkcji/procedury w językach proceduralnych dwoma elementami:

- metoda otrzymuje jako dodatkowy ukryty parametr obiekt, dla którego ma się wykonać. W kodzie metody będziemy się odwoływali do tego parametru za pomocą słowa “self”.
- W językach obiektowych występuje mechanizm metod wirtualnych, co oznacza, że decyzja o wyborze metody jest podejmowana podczas wykonania programu, a nie podczas kompilacji.
- Gdy obiekt otrzyma komunikat, reaguje w sposób zależny od swojej klasy. Jeśli w tej klasie jest metoda o nazwie takiej, jak nazwa komunikatu, wywołujemy ją, jeśli nie, to szukamy w nadklasie itd.

Metody wirtualne — przykład

Np. w programie zawierającym deklaracje klas:

```
class A {  
    procedure nazwa { write('A') }  
    procedure pisz {  
        write('To jest obiekt klasy '); self.nazwa  
    }  
}  
class B extends A {  
    procedure nazwa { write('B') }  
}  
var p:A;
```

wykonanie instrukcji:

```
p := new A; p.pisz
```

wypisze "To jest obiekt klasy A", a wykonanie:

```
p:= new B; p.pisz
```

wypisze "To jest obiekt klasy B".

Tablica metod wirtualnych

- Powszechnie stosowanym rozwiązaniem jest wyposażenie obiektu w tzw. tablicę metod wirtualnych, zawierającą adresy kodu odpowiednich metod.
- W językach z typami statycznymi, dopuszczalne komunikaty są znane podczas kompilacji. Można je ponumerować i reprezentować tablicę metod wirtualnych za pomocą zwykłej tablicy V , gdzie $V[k]$ zawiera adres metody, którą należy wykonać w odpowiedzi na komunikat numer k .
- wysłanie komunikatu k wymaga skoku ze śladem pod adres $V[k]$. Pozostałe elementy protokołu będą takie same, jak w przypadku zwykłych funkcji.
- Wybór metody zależy tylko od klasy obiektu; wszystkie obiekty danej klasy mogą mieć wspólną tablicę metod wirtualnych. W samym obiekcie umieszczamy jedynie adres tej tablicy.

Budowa tablic metod wirtualnych

- Budowa tablic metod wirtualnych oraz numerowanie komunikatów odbywa się podczas kompilacji, na etapie analizy kontekstowej.
- Tablice metod wirtualnych dla poszczególnych klas budujemy w kolejności przejścia drzewa dziedziczenia "z góry na dół".
- Tablica metod wirtualnych dla podklasy powstaje z tablicy dla nadklasy przez dodanie adresów metod zdefiniowanych w tej klasie.
- Jeśli metoda była już zdefiniowana "wyżej" w hierarchii, czyli jest redefiniowana, jej adres wpisujemy na pozycję metody redefiniowanej.
- Jeśli metoda pojawia się na ścieżce dziedziczenia pierwszy raz, jej adres wpisujemy na pierwsze wolne miejsce w tablicy metod wirtualnych.