

# Metody Realizacji Języków Programowania

## LLVM

Marcin Benke

MIM UW

10 października 2016

# Maszyna rejestrowa

## Rejestry

przechowują dane wewnątrz CPU

- + dostęp  $\simeq 10*$  szybszy niż do pamięci
- kosztowne; np x86 ma 7 rejestrów uniwersalnych

## LLVM

maszyna wirtualna, nieograniczona ilość rejestrów

generacja kodu na rzeczywisty procesor przez *alokację rejestrów*



Low Level Virtual Machine,  
<http://llvm.org/>

Narzędzia wspierające  
tworzenie kompilatorów

Biblioteki C++,  
ale także format tekstowy IR

Zainstalowane na `students`

# Historia

Początkowo projekt University of Illinois, 2002

Obecnie duży projekt OS, używany przez Apple, Nvidia, Adobe, Intel, etc.

CLANG — kompilator C/C++ do LLVM jest domyślnym kompilatorem na Mac OS X

W 2012 LLVM wyróżnione ACM Software Systems Award  
Wcześniej tę nagrodę uzyskały VMware, Make, Java, Apache,...

# Język LLVM IR

## Charakterystyczne cechy

- operacje postaci:

```
%t2 = add i32 %t0, %t1
```

- instrukcje są silnie typowane:

```
%t5 = fadd double %t4, %t3  
store i32 %t2, i32* %loc_r
```

- nowy rejestr dla każdego wyniku  
(tzw. Static Single Assignment, patrz dalsze wykłady)

## LLVM — przykład

```
declare void @printInt(i32) ; w innym module
define i32 @main() {
    %i1 = add i32 2, 2
    call void @printInt(i32 %i1)
    ret i32 0
}
```

---

```
$ llvm-as t2.ll
$ llvm-link -o out.bc t2.bc runtime.bc
$ lli out.bc
4
```

### Uwaga:

- nazwy globalne zaczynają się od @, lokalne od %
- nazwy zewnętrzne są deklarowane (@printInt)
- parametry funkcji w rejestrach

# Hello

```
declare i32 @puts(i8*)
@hello = internal constant [7 x i8]
        c"Hello\0A\00"

define i32 @main() {
    entry:
        %t0 = bitcast [7 x i8]* @hello to i8*
        %t1 = call i32 @puts(i8* %t0)
        ret i32 0
}
```

## Uwaga

Napis `@hello` jest stałą globalną

NB `\0A\00`

`@hello` jest rzutowane do typu `i8*`

## Hell<segmentation fault>

```
@hellostr = internal constant i32 u0x6c6c6548
declare i32 @puts(i8*)

define i32 @main() { entry:
    %t0 = bitcast i32* @hellostr to i8*
    %_ = call i32 @puts(i8* %t0)
    ret i32 0 }
```

```
> lli crash.bc
```

```
[1]      99879 segmentation fault  lli crash.bc
```



# Wartości bezimienne

Nienazwane wartości są numerowane tak, że ciało funkcji `main` możemy zapisać

```
bitcast [7 x i8]* @hellostr to i8*  
call i32 @puts(i8* %1)  
ret i32 0
```

co odpowiada

```
%1 = bitcast [7 x i8]* @hellostr to i8*  
%2 = call i32 @puts(i8* %1)  
ret i32 0
```

nienazwana etykieta otrzymała nazwę `%0`

# LLVM — silnia, rekurencyjnie

```
define i32 @fact(i32 %n) {  
    %c0 = icmp eq i32 %n, 0  
    br i1 %c0, label %L0, label %L1  
  
L0:  
    ret i32 1  
  
L1:  
    %i1 = sub i32 %n, 1  
    %i2 = call i32 @fact(i32 %i1)  
    %i3 = mul i32 %n, %i2  
    ret i32 %i3  
}
```

Uwaga:

- argumenty funkcji są deklarowane
- wszystko jest typowane, nawet warunki skoków
- skoki warunkowe tylko z “else”

# LLVM — typy (nie wszystkie)

- $n$ -bitowe liczby całkowite:  $i_n$ ,  $np$ :
  - `i32` dla `int`
  - `i1` dla `bool`
  - `i8` dla `char`
- nie ma podziału na liczby ze znakiem i bez;  
są operacje ze znakiem, `np`
  - `sle` — signed less or equal
  - `udiv` — unsigned div
- `float` oraz `double`
- `label`
- `void`
- wskaźniki:  $t^*$  (`np i8*` oznacza `char*`)
- tablice:  $[n * t]$  (uwaga: inny typ niż wskaźniki), `np`.  
`@hw = constant [13 x i8] c"hello world\0A\00"`
- struktury  $\{t_1, \dots, t_n\}$

# Zmienne globalne

```
static int a;  
int main() {  
    a = 42;  
    return a ;  
}
```

```
@a = internal global i32 0
```

```
define i32 @main() {  
    store i32 42, i32* @a  
    %1 = load i32* @a  
    ret i32 %1  
}
```

**Uwaga:** @a jest tak naprawdę typu i32\* (dlaczego?)

# Zmienne lokalne

```
int main() {  
    int a;  
    a = 42;  
    return a ;  
}
```

```
define i32 @main() {  
    %a = alloca i32  
    store i32 42, i32* %a  
    %1 = load i32* %a  
    ret i32 %1  
}
```

Instrukcja `alloca` przydziela pamięć w ramce wcielenia funkcji.

# LLVM — napisy

Typ `char*` z `C` to `i8*`

```
char* concat(char* s1, char* s2) {  
    char* t = malloc(strlen(s1)+strlen(s2)+1);  
    return strcat(strcpy(t,s1),s2); }
```

---

```
define i8* @concat(i8* %s1, i8* %s2) {  
    %1 = call i32 @strlen(i8* %s1)  
    %2 = call i32 @strlen(i8* %s2)  
    %3 = add i32 %1, 1  
    %4 = add i32 %3, %2  
    %5 = call i8* @malloc(i32 %4)  
    %6 = call i8* @strcpy(i8* %5, i8* %s1)  
    %7 = call i8* @strcat(i8* %6, i8* %s2)  
    ret i8* %7 }
```

## LLVM — napisy

Literały napisowe są tablicami [n x i8]

Trzeba je rzutować do i8\* np. przez bitcast

```
int main()    {  
    puts(concat("Hello,", " world\n"));  
    return 0; }
```

---

```
declare i32 @puts(i8*)  
declare i8* @concat(i8*, i8*)  
@s1 = private constant [7 x i8] c"Hello,\00"  
@s2 = private constant [8 x i8] c" world\0A\00"  
define i32 @main() {  
    %s1 = bitcast [7 x i8]* @s1 to i8*  
    %s2 = bitcast [8 x i8]* @s2 to i8*  
    %s3 = call i8* @concat(i8* %s1, i8* %s2)  
    %_ = call i32 @puts(i8* %s3)  
    ret i32 0 }
```

## LLVM — użycie

- `llvm-as` — asembler (do bitkodu)
- `llc` — generator kodu maszynowego (z bitkodu)
- `llvm-link` — linker (bitkodu)
- `lli` — interpreter (bitkodu)

```
llvm-as useconcat2.ll
llc -o useconcat2.s useconcat2.bc
llvm-as concat.ll
llc -o concat.s concat.bc
gcc -o useconcat useconcat2.s concat.s
```

### Alternatywnie:

```
llvm-as useconcat2.ll
llvm-as concat.ll
llvm-link -o out.bc useconcat2.bc concat.bc
lli out.bc
```



# Clang

## Kompilator C do LLVM

```
int puts(char*);  
int main() { puts("Hello\n"); return 0; }
```

---

```
clang -O0 -o chello.ll -emit-llvm -S chello.c
```

---

```
@.str = private constant [7 x i8] c"Hello\0A\00", align 1  
  
define i32 @main() #0 {  
    %1 = alloca i32, align 4  
    store i32 0, i32* %1  
    %2 = call i32 @puts(i8* getelementptr  
                        inbounds ([7 x i8]* @.str, i32 0, i32 0))  
    ret i32 0  
}
```

getelementptr — arytmetyka wskaźników

inbounds — ze sprawdzeniem, że wynik w granicach  
zaalokowanej pamięci

## LLVM — tablice

```
void print7(char a[]) {  
    puts(&a[6]);  
}  
int main(){ print7("Hello world!\n");}
```

```
define void @print7(i8* %a) {  
    %x = getelementptr i8* %a, i64 6  
    %r = call i32 @puts(i8* %x)  
    ret void  
}  
define i32 @main() {  
    %x = getelementptr [14 x i8]* @s, i64 0, i32 0  
    call void @print7(i8* %x)  
    ret i32 0  
}
```

**Uwaga:** `getelementptr` tylko oblicza adres, nie czyta pamięci.

# Dlaczego dwukrotnie 0?

```
@.str = private constant [7 x i8] c"Hello\0A\00", align 1

define i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1
    %2 = getelementptr inbounds [7 x i8]* @.str, i32 0, i32 0
    %3 = call i32 @puts(i8* %2)
    ret i32 0
}
```

## LLVM to nie jest C

@.str jest naprawdę typu [7 x i8]\*

## Dlaczego dwukrotnie 0?

```
struct Pair { int x, y; };
```

```
int f(struct Pair *p) {  
    return p[0].y + p[1].x;  
}
```

```
define i32 @f(%Pair* %p) {  
    %1 = getelementptr %Pair*, i64 0, i32 1  
    %2 = load i32*, %1  
    %3 = getelementptr %Pair*, i64 1, i32 0  
    %4 = load i32*, %3  
    %5 = add i32 %4, %2  
    ret i32 %5  
}
```

## LLVM — tablice i struktury

```
struct list {  
    char hdr[16];  
    int car;  
    struct list* cdr;  
};  
int foo(struct list a[]) {  
    return a[7].car;  
}
```

---

```
%struct.list = type { [16 x i8],  
                      i32, %struct.list* }  
define i32 @foo(%struct.list* %a) {  
    %1 = getelementptr %struct.list* %a, i32 7, i32 1  
    %2 = load i32* %1  
    ret i32 %2  
}
```

### Definicja

*Blok prosty jest sekwencją kolejnych instrukcji, do której sterowanie wchodzi wyłącznie na początku i z którego wychodzi wyłącznie na końcu, bez możliwości zatrzymania ani rozgałęzienia wewnątrz.*

Kod LLVM:

- etykietowane bloki proste
- każdy blok kończy się skokiem (`ret` lub `br`)
- nie ma przejścia od ostatniej instrukcji bloku do pierwszej kolejnego (bloki można permutować)
- skoki warunkowe mają dwa cele:  
`br i1 %c0, label %L0, label %L1`
- nie można do skoczyć do bloku wejścia do funkcji
- `call` nie kończy bloku.

# Silnia, naiwnie (niepoprawnie)

```
define i32 @nfac(i32 %n) {  
    %r = 1  
    %i = n  
loop:  
    %c = icmp sle i32 %i, 1  
    br i1 %c, label %end, label %body  
body:  
    %r = mul i32 %r, %i  
    %i = sub i32 %i, 1  
    br label %loop  
%end:  
    ret i32 %r    }
```

## Błędy

- istotny: po dwa przypisania na %i i %r
- średni: blok przed loop nie kończy się skokiem
- drobny: w LLVM nie ma instrukcji `reg = val`

## LLVM — silnia, iteracyjnie

```
; r = 1
; i = n
; while (i > 1):
;   r = r * i
;   i = i - 1
; return r
define i32 @fact(i32 %n) {
entry:
; zmienne lokalne, alokowane na stosie
    %loc_r = alloca i32
    %loc_i = alloca i32
; r = 1
    store i32 1, i32* %loc_r
; i = n
    store i32 %n, i32* %loc_i
    br label %L1
```



## LLVM — silnia, iteracyjnie

```
; while i > 1:
L1:    %tmp_i1 = load i32* %loc_i
        %c0 = icmp sle i32 %tmp_i1, 1
        br i1 %c0, label %L3, label %L2

; cialo petli
L2:
; r = r * i
        %tmp_i2 = load i32* %loc_r
        %tmp_i3 = load i32* %loc_i
        %tmp_i4 = mul i32 %tmp_i2, %tmp_i3
        store i32 %tmp_i4, i32* %loc_r

; i = i-1
        %tmp_i5 = load i32* %loc_i
        %tmp_i6 = sub i32 %tmp_i5, 1
        store i32 %tmp_i6, i32* %loc_i
        br label %L1
```

## LLVM — silnia, iteracyjnie

```
; po petli:  
; return r
```

L3:

```
    %tmp_i8 = load i32*, %loc_r  
    ret i32 %tmp_i8
```

```
}
```

# Optymalizator

```
> opt -mem2reg ifac.ll > regfac.bc  
> llvm-dis regfac.bc
```

```
define i32 @fact(i32 %n) {  
entry: br label %L1  
L1:  
    %i.1 = phi i32 [%n, %entry], [%i.2, %L2]  
    %r.1 = phi i32 [1, %entry], [%r.2, %L2]  
    %c0 = icmp sle i32 %i.1, 1  
    br i1 %c0, label %L3, label %L2  
L2:  
    %r.2 = mul i32 %r.1, %i.1  
    %i.2 = sub i32 %i.1, 1  
    br label %L1  
L3:  
    ret i32 %r.1  
}
```

## Funkcje $\phi$

Analizy i przekształcenia kodu są łatwiejsze jeśli każda zmienna ma tylko jedną definicję.

Taką postać nazywamy postacią SSA: Static Single Assignment — **statycznie** na każdą zmienną jest tylko jedno przypisanie (nic nie stoi natomiast na przeszkodzie by wykonało się wiele razy, np. w pętli)

LLVM wymaga kodu w postaci SSA

Używamy "funkcji"  $\phi$  (od "phony"), która wybiera jedną z wartości, w zależności skąd sterowanie weszło do bloku.