Sequence 4.2 – LLVM Intermediate Representation

P. de Oliveira Castro S. Tardieu

LLVM IR

- Design Principles
  - Human readable
  - Represent high-level languages cleanly
  - Abstracts functions and function calls
  - Flat language composed of *LLVM Instructions*
    - Abstract low level assembly
    - Every value and instruction is *typed*

LLVM Types

LLVM defines **first class** (basic) types and *derived* types. Example:

Type	Meaning
void	missing value
i1	1 bit integer (boolean)
i8	8 bits integer (char)
i32	32 bits integer (int)
float	32 bits floating point
*i32**	pointer to an int
void (i32)	function taking an integer and returning void
{ i8, i32 }	structure with two fields: a char and an int

LLVM Abstracts functions and function calls

The simple tiger program,

```
print_int(42)
```

produces the following LLVM IR,

```
define i32 @main() {                    ; defines the main function
                                        ; returning an 32 bit int

  call void @__print_int(i32 42)        ; call to a primitive
                                        ; returns void

  ret i32 0                             ; main returns 0
}
```

Values

LLVM instructions with non-void return type can be assigned to a *value*

```
<result> = add <type> <left>, <right>
```

```
%result = add i32 3, 2
```

- Values start with a % symbol
- Values can only be assigned to once, but can be read many times
- Here `%result` is a 32 bit integer and evaluates to 5

Memory locations

The `alloca` instruction allocates memory space on the stack and returns a pointer

```
%ptr = alloca i32        ; %ptr is of type i32*
```

`store` and `load` instructions read from and write to a pointer

```
; Write 5 into the pointer. In C, *ptr = 5;
store i32 5, i32* %ptr

; Read the pointer content. In C, int content = *ptr;
%content = load i32, i32* %ptr
```

Local variables in LLVM

The simple way: `alloca` reserves room on the stack for the local variable, `store` and `load` accesses its content.

```
let function f(a: int, b: int): int =
  let var c := a + b in c end
in ... end
```

```
define i32 @f(i32, i32) #0 {
  %c = alloca i32                    ; %c <- address of c
  %3 = add i32 %0, %1                ; %3 <- a + b
  store i32 %3, i32* %c              ; Store %3 into c
  %4 = load i32, i32* %c             ; Load c into %4
  ret i32 %4                         ; and return it
}
```

Simplification

Using `alloca` for all local variable does not seem to lead to efficient code. Fortunately, a later optimization pass called *mem2reg* (memory to register) will remove all useless `alloca`, `store` and `load`, and replace them with direct virtual register manipulation:

```
define i32 @f(i32, i32) #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}
```

which will give, in ARM thumb assembly:

```
f:
    add r0, r1
    bx  lr
```

Branches and Labels

Code locations can be represented with *labels*. The branch `br` instruction jumps to another location.

```
; unconditional jump to label here
br label %here

here:
  ; conditional jump depends on the %condition value
  br i1 %condition, label %when_true, label %when_false

when_true:
  ...

when_false:
  ...
```

Integer comparison

Integer comparison `icmp` operation returns an `i1`

```
%result = icmp ne i32 %a, %b ; true when %a <> %b

%result = icmp eq i32 %a, %b ; true when %a == %b

%result = icmp sge i32 %a, %b ; true when %a >= %b

etc...
```