



## Sequence 5.1 – Building stack frames in LLVM

---

P. de Oliveira Castro   S. Tardieu

## Reminder: Stack frames

We have seen earlier that:

- A function can access its local variables and parameters (a function parameter as seen from the function is no different than a local variable).
- A function can access local variables defined outside the function (escaping local variables) if they are lexically visible, as well as outer function parameters.
- Function parameters and escaping local variables are stored in a *frame*, which is itself stored onto the stack. This is also called the *function stack frame*.
- Every frame contains a pointer to the stack frame one level up.
- This pointer is passed as an extra parameter to every function.

Do not hesitate to visualize sequence 3.3 again should you need to refresh your memory.

# Building stack frames in LLVM

We can use a simple yet powerful model in LLVM to build stack frames:

- For every function, we declare a new compound type containing:
  - Every parameter received by the function (including the upper stack frame pointer).
  - Every escaping local variable created in this function.
- The appropriate stack frame pointer is passed as the first argument of every function call.
- At function entry, a new stack frame object of the proper compound type is allocated using `alloca` and function parameters are copied to this object.
- Escaping local variables are now fields of this frame object.
- Non-escaping local variables are not stored in the frame.

## Isn't that costly though?

- Creating an object using `alloca`, copying the parameters into it, and forcing escaping local variables to be fields of this object residing in physical memory has a cost.
- This cost is the price to pay to get access to those parameters and escaping local variables from an inner function.

However, the `mem2reg` optimization pass of LLVM will take care of removing the extra copies if we do not need to pass a pointer to this frame object to another function, in which case there will be no cost.

## Could we optimize it further?

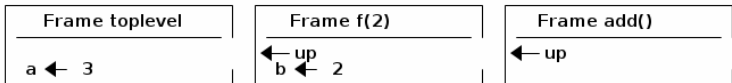
In a production-quality compiler, some optimizations would be implemented. For example:

- If a function does not call other functions, it does not need to build a frame object. This is already taken care of in the `mem2reg` optimization pass of LLVM. Such a function is called a *leaf function*.
- If a function does not use any outer variable and is a leaf function, it does not need to receive the outer frame pointer. Let us call it a *pure function*.
- If a function does not use any outer variable and only call pure functions, then it is itself a pure function and does not need to receive the outer frame pointer.

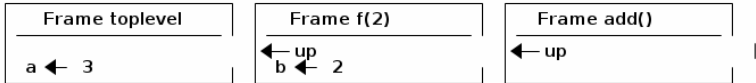
This list of optimizations is in no way exhaustive.

## Visualizing a frame (from sequence 3.3)

```
let var a := 3
  function f(b: int) =
    let function add() = a := a + b
    in add() end
in
  f(2); print_int(a)  /* Prints 5 */
end
```



# Visualizing frame types in LLVM



We build LLVM types named `ft_` (for *frame type*) followed by the full path of the function. Here is the corresponding LLVM IR:

```
%ft_main = type { i32 }           # { a }  
%ft_main.f = type { %ft_main*, i32 } # { up, b }  
%ft_main.f.add = type { %ft_main.f* } # { up }
```

Except for `main`, which has no frame `up` since it represents the toplevel, every frame starts with a pointer to an object of the lexically outer frame type (called *static link*).

# Building the frame in LLVM

Here is the IR code for the function `f` defined previously:

```
define void @main.f(%ft_main* %sl, i32 %b) {
  # Allocate frame object
  %frame = alloca %ft_main.f
  # Store outer frame pointer (%sl) at first position in %frame
  %0 = getelementptr inbounds %ft_main.f, %ft_main.f* %frame,
    i32 0, i32 0
  store %ft_main* %sl, %ft_main** %0
  # Store %b parameter at second position in %frame
  %1 = getelementptr inbounds %ft_main.f, %ft_main.f* %frame,
    i32 0, i32 1
  store i32 %b, i32* %1
  # Call add function and give it a pointer to the frame object
  call void @main.f.add(%ft_main.f* %frame)
  ret void
}
```



# Conclusion

- A frame compound type is defined for every function.
- The frame contains a slot for the outer frame pointer (except at top-level), every parameter, and every escaping local variable.
- At function entry, the frame is allocated and the outer frame pointer (also called static link) and function parameters are copied into the frame.
- Escaping local variables become fields of the frame object.