Sequence 3.3 – Stack frames

P. de Oliveira Castro S. Tardieu

Accessing variables

- The binder has bound every variable use (`Identifier` node in the AST) to its declaration (`VarDecl` node).
- However, because of recursive (or mutually recursive) functions, a variable declaration (including a function parameter) may denote several values at the same time.

```
1  let function fact(n: int): int =
2      if n <= 1 then 1 else n * fact(n-1)
3  in
4   fact(5)
5  end
```

- Here we end up with 5 nested calls to `fact`, with 5 values for `n` (5 for the first call, then 4, . . . , down to 1).

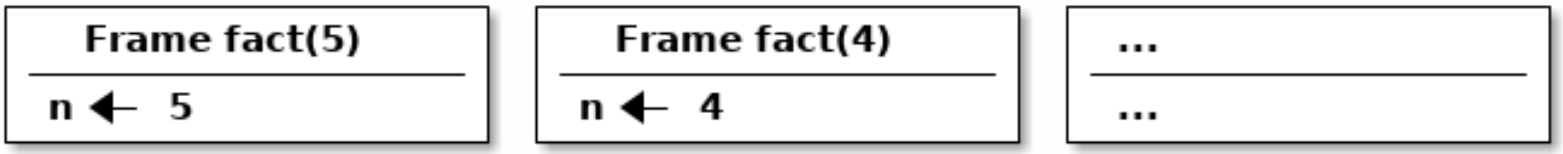
Function frames

- Every time we enter a new function, we need to build a new *frame* to hold the values of the variables defined in the function (starting with the function parameters).
- Frames must not be confused with scopes: scopes are used by the binder to lexically determine which variable must be visible, and frames are used to hold the variables contents at run time.
- Conceptually, a frame is a mapping between `VarDecl` nodes and values representing the variable content.
- Frames are stored on the stack.

Back to our example

```
let function fact(n: int): int =
      if n <= 1 then 1 else n * fact(n-1)
in
  fact(5)
end
```

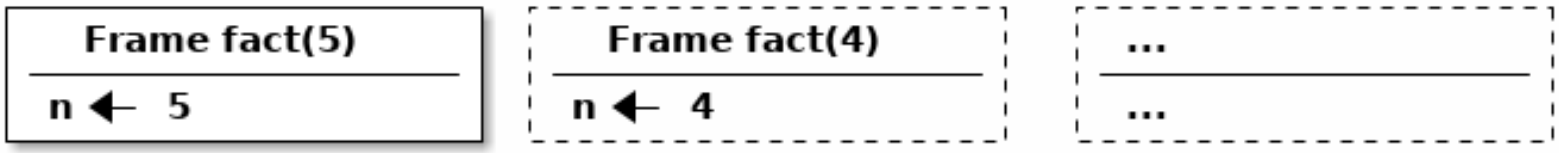
- During the first call `fact(5)`, a frame is built on the stack and associates the value 5 to the `VarDecl` of `n`.
- During the second call `fact(n-1=4)`, a frame is built on the stack and associates the value 4 (result of the expression) to the `VarDecl` of `n`.



Discarding stack frames

```
let function fact(n: int): int =
      if n <= 1 then 1 else n * fact(n-1)
in
  fact(5)
end
```

- When we later get back from this second call with a result of `fact(4)=24`, we discard the frame built for this call from the stack, which gives us again access to `n` being equal to 5. `n*24` is equal to 120, which is the expected result.



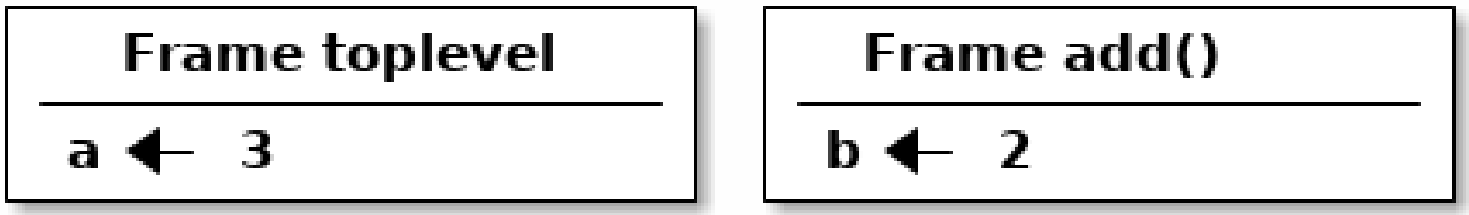
Putting everything together

- The binder uses scopes to bind every `Identifier` node in the AST to the corresponding `VarDecl` node.
- At run time, every function call will create a new frame in which we will bind newly encountered `VarDecl` to a place where the corresponding value will be stored.
- Reading or modifying a variable value at run time is easy: just look it up using its `VarDecl` in the current frame.
- However, only locally created variables are present in the current frame. How is it possible to access variables declared in an outer scope?

Accessing outside variables

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

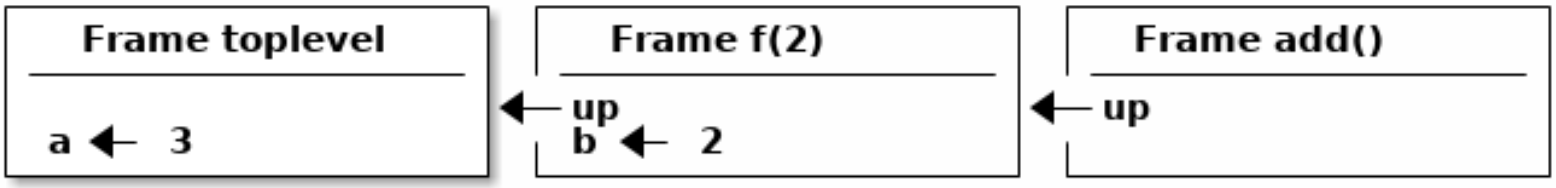
- A toplevel frame contains the value of variable a
- When add is called, a new frame is created and holds the value for the b parameter.
- It is necessary to access the one level up frame from within add.
- We need to pass add an extra parameter which is a pointer onto the outer frame.



Accessing variables more than one level above

```
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
```

- When accessed from function add, a is located in the frame two level up.
- In every frame, we will store an up pointer to the frame above. This way, the code generated for add will be able to walk the frames chain up two times in order to locate a.



Knowing how far to walk the frames chain up

- In order to know how many times one must walk the frame chain up, we assign *depth* information to every variable declaration and use in the binder.
- The starting depth is 0. Every time we encounter a function declaration and analyze its body, we increment depth by one. When we are done analyzing the body, the depth is decremented.
- Every time we encounter a variable declaration (`VarDecl`) or a variable use (`Identifier`), we store its depth.
- The difference of depths between a variable use (`Identifier`) and a variable declaration (`VarDecl`) gives the number of steps we must walk up the frames chain to find the variable content.
- If a variable is accessed from a depth greater than its declaration, it is said to *escape* its frame.

Annotated example

- /*e*/ denotes an escaping declaration, /*2*/ that the depth difference is 2. Notice that b escapes too, it is accessed from within add while being declared in f.

```
let var a/*e*/ := 3
    function f(b/*e*/: int) =
      let function add() = a/*2*/ := a/*2*/+b/*1*/
      in add() end
in
  f(2); print_int(a)  /* Prints 5 */
end
```

- f is given the outer frame as extra parameter and stores it in its own frame, and add is given f frame as extra parameter and stores it in its own frame. Walking up is easy then.

Sibling functions

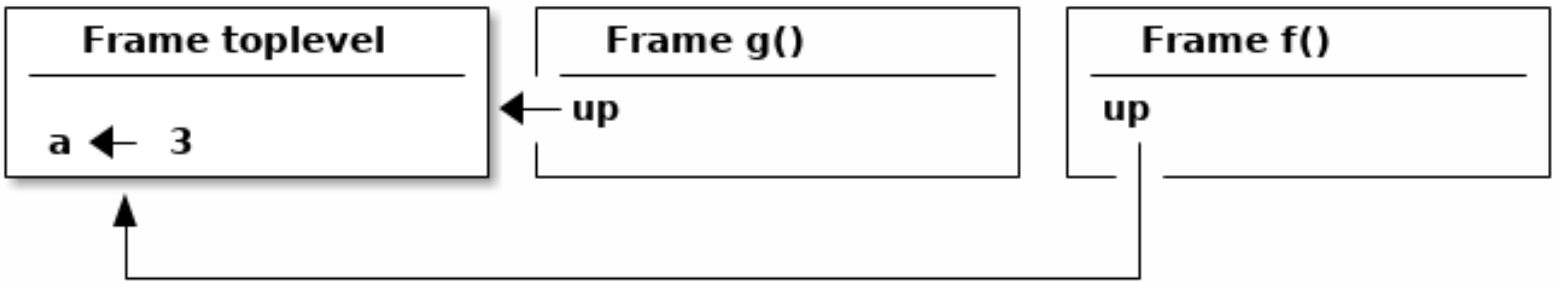
- It is important to understand that the extra parameter denotes the frame corresponding to the level right above the called function, which is not necessarily the current frame.

```
let var a := 3
    function f(): int = a + a
    function g(): int = f() + f()
in
  print_int(g())  /* Prints 12 */
end
```

- g is passed the outermost frame as extra parameter.
- When g calls `f`, it needs to pass it the outermost frame as well, and not its own frame. `f` and g are declared at the same depth and thus share the same extra enclosing frame parameter.

Visualizing the sibling stack frames

```
let var a/*e*/ := 3
    function f(): int = a/*1*/ + a/*1*/
    function g(): int = f() + f()
in
  print_int(g())  /* Prints 12 */
end
```



Choosing the extra frame parameter to pass

- When calling a function declared at the same depth as the depth where the call takes place, the current frame must be passed as extra parameter, since the function body depth will be exactly one more than the current depth.
- When calling a function declared k levels above the depth of the call (*e.g.*, 1 for a sibling function, since the call is within the body), the frames chain must be walked up k times.

For example, a recursive call to the current function will walk the frames chain one time (a function is its own sibling). As a consequence, it will give itself the same frame that it has received.

Conclusion

- Scopes are used by the binder to associate a variable use (`Identifier`) to a variable declaration (`VarDecl`).
- Frames are used at run time as a container for the variables content.
- Every function receives a pointer to the enclosing function frame and stores it in its own frame. This pointer is necessary to access variables defined outside the function but visible from within.
- By going up the stored frames chain, a function may access a variable defined several levels above.