

Activation Records (AR)

Tobias Lamote & Tim Robensyn

9 november 2023

Inhoudstafel

- ▶ Herhaling
- ▶ Parameter passing
- ▶ Tail call optimization
- ▶ Stack smashing
- ▶ Demo met gdb
- ▶ Nested functions
- ▶ Continuations
- ▶ Examenvragen

Situering

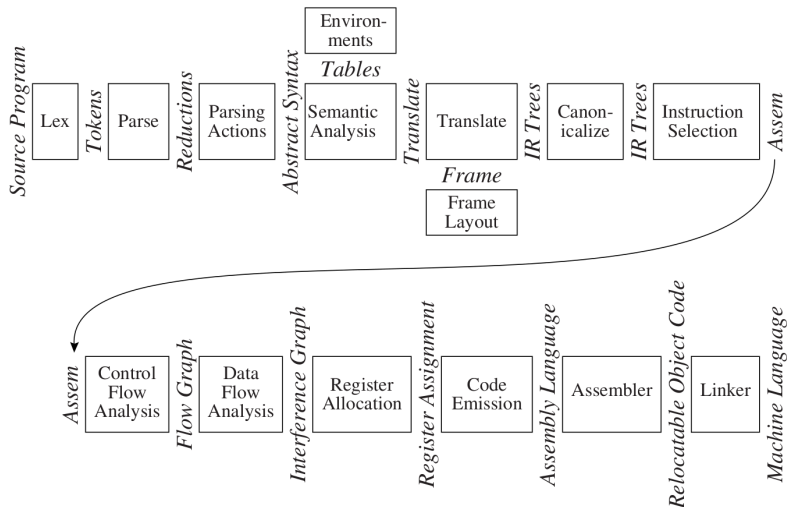
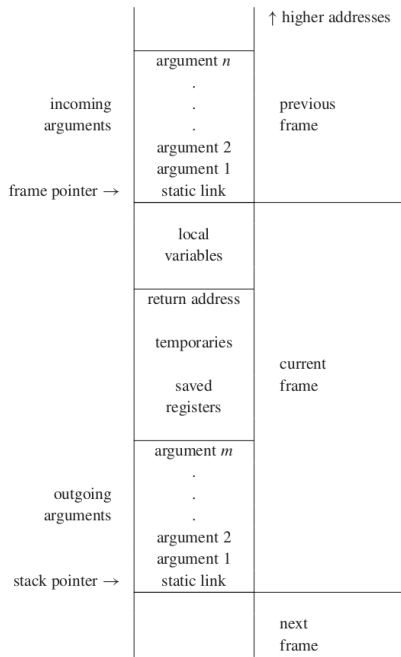


FIGURE 1.1. Phases of a compiler, and interfaces between them.

Wat zit er in een stack frame?

Stack van het handboek



- ▶ Geen echte *stack*
- ▶ Geen *echte* stack
- ▶ Register allocator (zie later met Jacob)

Procedure linkages

The linkage divides responsibility between *caller* and *callee*

	Caller	Callee
Call	<i>pre-call</i> <ol style="list-style-type: none">1. allocate basic frame2. evaluate & store params.3. store return address4. jump to child	<i>prologue</i> <ol style="list-style-type: none">1. save registers, state2. store FP (dynamic link)3. set new FP4. store static link5. extend basic frame (for local data)6. initialize locals7. fall through to code
Return	<i>post-call</i> <ol style="list-style-type: none">1. copy return value2. deallocate basic frame3. restore parameters (if copy out)	<i>epilogue</i> <ol style="list-style-type: none">1. store return value2. restore state3. cut back to basic frame4. restore parent's FP5. jump to return address

At compile time, generate the code to do this

At run time, that code manipulates the frame & data areas

Nut van een frame pointer

- ▶ Frames van variabele lengte (dynamic link)
- ▶ Compiler kan vroeger aannames maken
- ▶ Programmeur kan functies volgen in assembly

Parameter passing

Waarom geven we parameters zo veel mogelijk door via registers?

Parameter passing

Waarom geven we parameters zo veel mogelijk door via registers?

Snelheid!

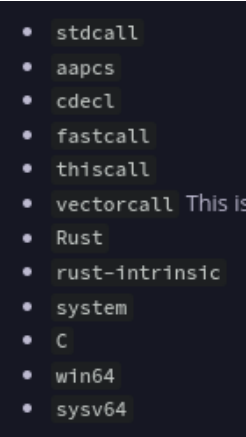
1. leaf procedures: $l = intern * (children - 1) + 1$
2. interprocedural register allocation
3. dead variables
4. specific architectures (register windows)

Calling conventions

- ▶ iedere taal/compiler kan kiezen
- ▶ samenwerken: standaard gebruiken
- ▶ stack data layout
- ▶ caller-save vs callee-save
- ▶ leaf functies gebruiken caller-save

ABI

- ▶ compiler kan verschillende ABIs ondersteunen
- ▶ vb. rust



```
• stdcall
• aapcs
• cdecl
• fastcall
• thiscall
• vectorcall This is
• Rust
• rust-intrinsic
• system
• C
• win64
• sysv64
```

- ▶ windows 32 c: stdcall
- ▶ windows 64 c: C

Tail call optimization

- ▶ recursive functies
- ▶ stack herbruiken

Tail call optimization

```
int factorial(int n) {  
    if (n == 1)  
        return 1;  
    return n * factorial(n-1);  
}  
  
void main() {  
    int fac = factorial(5);  
}
```

```
int factorial(int n, int accumulator) {  
    if (n == 1)  
        return accumulator;  
    return factorial(n-1, accumulator*n);  
}  
  
void main() {  
    int fac = factorial(5, 1);  
    printf("%d", fac);  
}
```

Stack Smashing

What can the compiler do?

- ▶ Add stack canaries (terminator, random, random XOR'ed)
- ▶ Add bounds checking

Meer in de les Development of Secure Software.

Demo: spelen met gdb

Examples:

- ▶ https://github.com/Toobsterrr/activation_records.git
- ▶ Functie die andere functie callt
- ▶ Functie met veel parameters (om spilling van de registers te demonstreren)
- ▶ Functie die waardes returnt
- ▶ Functie die via een buffer de stack manipuleert
- ▶ Verschillende optimization levels testen
- ▶ Tail call optimization

Nested functions

```
1      type tree = {key: string, left: tree, right: tree}
2
3      function prettyprint(tree: tree) : string =
4      let
5          var output := ""
6
7          function write(s: string) =
8              output := concat(output,s)
9
10         function show(n:int, t: tree) =
11             let function indent(s: string) =
12                 (for i := 1 to n
13                     do write(" ");
14                     output := concat(output,s); write("\n"))
15                 in if t=nil then indent(".")
16                     else (indent(t.key);
17                         show(n+1,t.left);
18                         show(n+1,t.right))
19             end
20
21         in show(0,tree); output
22     end
```

PROGRAM 6.3. Nested functions.

Nested functions: solutions

- ▶ Static links
- ▶ Display
- ▶ Lambda lifting

Wanneer is een stack niet genoeg?

Nested functions && Functions as return values

Oplossing

Heap-allocation en garbage collection

Continuations

Definitie

A continuation is the abstract concept represented by the control stack, or dynamic chain of activation records, in a typical programming-language implementation.

- ▶ Zoals closures environments opslaan, zo slaan continuations de huidige controlecontext op (de stack)
- ▶ Gelijkenissen met multithreading
- ▶ Kan gebruikt worden om try-catch structuren te maken

Continuations: implementatie

- ▶ The garbage-collection strategy
- ▶ The spaghetti stack

Examenvragen

- ▶ Leg uit hoe de stack verandert wanneer een nieuwe procedure wordt opgeroepen.
- ▶ Leg uit hoe het doorgeven van parameters via de registers zorgt voor efficiënter geheugengebruik.
- ▶ Leg tail call optimization uit en pas het toe op een voorbeeldprogramma.
- ▶ Gegeven een voorbeeldprogramma, kan dit programma activation records gebruiken die toegewezen zijn op een stack? Waarom?

Extra source

- ▶ Laatste hoofdstuk van deze slides:
<https://web.cs.ucla.edu/~palsberg/course/cs132/lec.pdf>
- ▶ *Implementation strategies for First-Class Continuations* van William D. Clinger (voor de geïnteresseerden)