

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

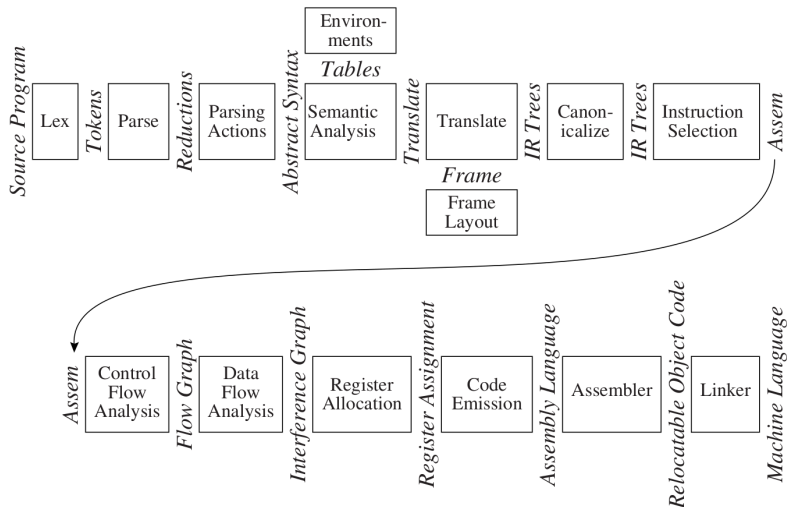
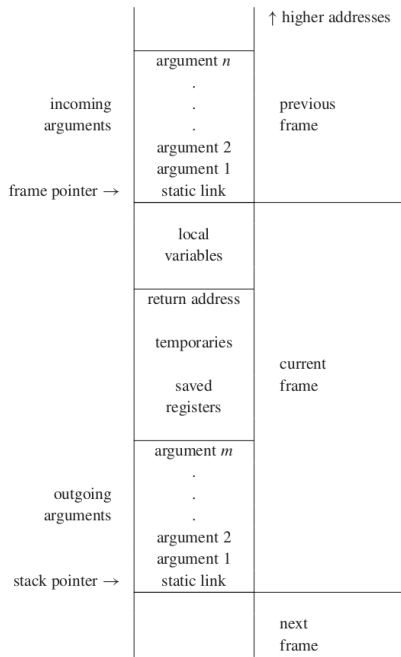


FIGURE 1.1. Phases of a compiler, and interfaces between them.

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!

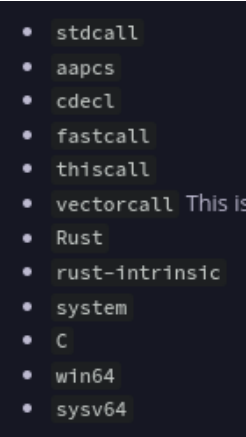
1. leaf procedures moeten vaak niet naar de stack schrijven:
$$l = intern * (children - 1) + 1$$
2. interprocedural register allocation: vermijden van conflicten tussen functies
3. dead variables: onnodige variabelen mogen overschreven worden
4. specific architectures: (vb. register windows: elke functie andere registers)

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

- ▶ recursieve functies
- ▶ stack herbruiken
- ▶ laatste call moet de functie zelf zijn

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

functions with parameters

```
void second(int vals[]) {  
    vals[0] = 8;  
    vals[3] = 9;  
}
```

```
void first(int x, int y, int z, int a, int b, int c, int s) {  
    int arr[] = {x,y,6,7};  
    second(arr);  
}
```

```
void main() {  
  
    int x = 2;  
    int y = 3;  
    first(x, y, 9, 8, 7, 6, 4);  
    int z = x + y;  
  
}
```


functions with return value

```
int* second() {  
    int* arr = malloc(3*sizeof(int));  
    arr[0] = 2;  
    arr[1] = 3;  
    arr[2] = 4;  
    return arr;  
}
```

```
int first() {  
    int* arr = second();  
    return 1;  
}
```

```
void main() {  
  
    int i = first();  
  
}
```

buffer overflow vulnerability

```
#include <stdio.h>

void forbidden_function()
{
    int bad[4] = {1, 2, 3, 4};
}

void okay_function()
{
    char string[5] = "baaa";
    string[21] = 'I';
    // string[21] = 'i';
    string[22] = 'Q';
}

void main ()
{
    okay_function();
}
```

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

Probleem

Hoe raakt een *inner* function aan de lokale variabelen van zijn *outer* function?

Free variables

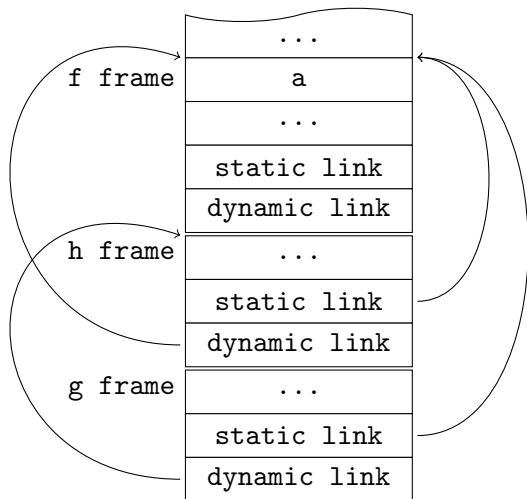
De variabelen van outer functions gebruikt in een lokale functie worden *free* (vrij) genoemd omdat ze niet *bound* (gebonden) zijn door de definitie van de lokale functie.

Nested functions: static links

Voeg een *static link* toe in elk frame. De link verwijst naar het frame van de outer function. Vrije variabelen kunnen nu via pointer dereferencing bereikt worden vanuit inner functions.

Static links: voorbeeld

```
f() {  
  a = 1;  
  g() {  
    a += 1;  
  }  
  h() {  
    g();  
  }  
  h();  
  return a;  
}
```



Nested functions: displays

- ▶ Nadeel static links: veel nesting leidt tot veel pointer dereferencing
- ▶ Idee: gebruik globale datastructuur!

Nested functions: lambda lifting

Idee:

Vermijd het probleem door vrije variabelen weg te werken

1. Geef alle functies globaal unieke namen
2. Voor elke lokale functie:
 - 2.1 Voeg alle gebruikte vrije variabelen toe als parameters zodat alle variabelen gebonden zijn
 - 2.2 Verplaats de lokale functie naar de globale scope

Lambda lifting: voorbeeld

Before:

```
f() {  
    a = 1;  
    g() {  
        a += 1;  
    }  
    h() {  
        g();  
    }  
    h();  
    return a;  
}
```

After:

```
g(a) {  
    return a + 1;  
}  
h(a) {  
    return g(a);  
}  
f() {  
    a = 1;  
    return h(a);  
}
```

Wanneer is een stack niet genoeg?

Voorwaarden

Nested functions && Functions as return values

→ Als functies met vrije variabelen langer kunnen leven dan hun outer functions is er een probleem aangezien de vrije variabelen dan kunnen verdwijnen wanneer het frame van de outer function van de stack wordt 'gepopt'.

Oplossing: closures

Closure

Een closure bestaat uit de functie zelf **plus** een voorstelling van de lexicale omgeving (environment) van die functie op het moment dat die gecreëerd werd.

Implementatie

De stack is geen optie (zie vorige slide) dus moet er gebruik worden gemaakt van de heap. Dit kan met garbage collection om closures die niet meer gebruikt worden veilig te verwijderen.

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

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)