# Compiler Construction

## Run-time systems (exekveringssystem)

Lennart Andersson

Revision 2012-02-15

2012



## Recursive computation

An extremely inefficient method for computing Fibonacci numbers:

```
static long f(int n) {
  long f1, f2;
  if (n<=1) {
    return 1;
  }
  f1 = f(n-1);
  f2 = f(n-2);
  return f1+f2;
}
```

$$f(0) = 1$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2),\ n \geq 2$$

How much time is required to compute $f(n)$?

How much memory?



## Global variables

```
void p1() {
  int i1=0;
  void p2() {
    i1++;
  }
  p2();
}
```

How can p2 access i1?



## Global variables with recursion

```
void p1() {
  int i1=0;
  void p2(int i) {
    if (i>0) {
      i1++;
      p2(i-1);
    }
  }
  p2(3);
}
```

How can p2 access i1?

How is the argument transferred to p2?

## Return values

```
void p1() {
  int i1=0;
  int p2(int i) {
    if (i>0) {
      i1++;
      return p2(i-1);
    } else {
      return i1;
    }
  }
  i1 = p2(3);
}
```

How are return values passed back?



## Run-time systems

How do we organize block instances during execution?

- global variables and constants (instance of the "outermost" block)
- activations (method instances)
- objects (class instances)
- ...

Method calls

- How are calls and returns handled?
- How are parameters transmitted?

Variables

- How are local / non-local variables accessed?

Support for object-oriented languages

- inheritance, overriding, dynamic dispatch, garbage collection



## How can data be stored in the machine?

Registers

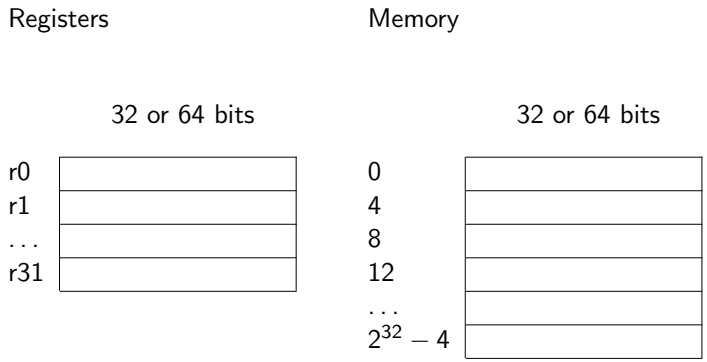
- small number (typically 8-32)

Memory

- large (in the order of Mbyte or Gbyte)
- write-protected area (typically for storing the code)
- read/write area (typically for storing data)

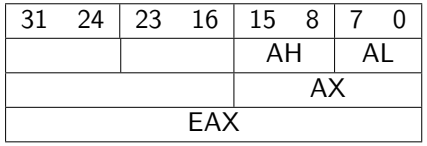


## Registers and memory



## Intel 386 registers

The Intel processor has eight 32 bit general registers. Four of them have a special structure, A, B, C and D. This is the A register.




Different parts of the register have different names.



## Typical use of the machine

Registers

- A few registers are dedicated to specific functions in the run-time system, e.g., program counter, stack pointer, etc.
- The rest of the registers are used for address calculations, temporary results in computations, parameter transmission etc.

Memory

- Statically allocated data
- Stack for dynamic allocation of activations (method instances)
- Heap for dynamic allocation of objects (class instances)
- Code (usually in a write-protected area)



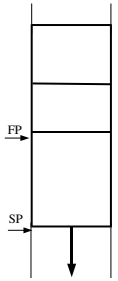
## Stack of activation records

An area in memory is reserved for the stack

- each method activation is represented by an *activation record* or *frame* for storing local variables, etc.
- stack grows downward in illustration

Typical dedicated registers

- FP — Frame Pointer — points to the first word of the current frame
- SP — Stack Pointer — points to the first unused word in the stack
- PC — Program Counter — points to the current instruction in the code (not shown here)






## Synonyms

activation record

activation

stack frame

frame

Swedish: aktiveringspost

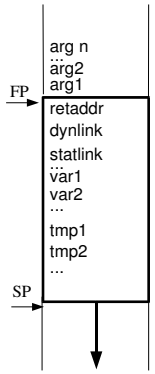
## Example of activation record layout, Intel style

Head

- retaddr — Return Address — points to the instruction in the code where the calling activation should continue execution when the current activation returns
- dynlink — Dynamic Link — points to the calling activation
- statlink — Static Link — points to the activation for the enclosing block

Data

- $arg_i$ — area for an argument
- $var_j$ — area for a local variable
- $tmp_k$ — area for a temporary variable
- the A register is used for the return value





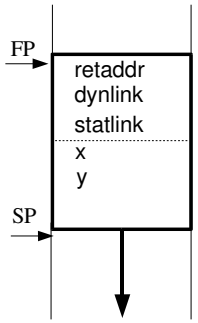

## Example

```
CODE                    STACK
void p1() {             p1-> return address
  int x = 1;                 dynamic link
  int y = 2;                 static link
  void p2() {             x  1
    int z = y+1;          y  2
    p3();               p2-> return address in p1
  }                            dynamic link -> p1
  void p3(){                   static link  -> p1
PC->int t = x+3;          z  3
  }                     p3-> return address in p2
                               dynamic link -> p2
  p2();                        static link  -> p1
  y++;                    t
}
```



## Access to local variables

```
int p() {
  int x = 1;
  int y = 2;
  y++;
  return x+y;
}
```




What is the address of y?
yAddr = FP+hSize+offset(y)
where

hSize is the size of the head (3 words in this layout)
offset(y) is the offset for variable y, counting from the start of the data part of the activation (1 word in this case)

We can access y at M[yAddr] where M[] is the memory, viewed as an array of words.



## Example of assembly code

Source:

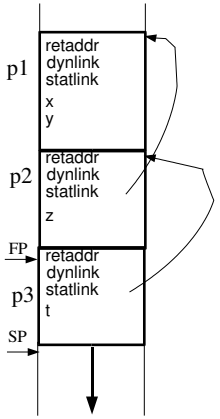
```
y++;
```

Assembly, assuming byte-addressed memory and 32 bit words:

```
ADD FP 16 R1  // R1 := byte address for y (FP + 4*(3+1))
LOAD R1 R2    // R2 := y
INC R2        // R2++
STORE R2 R1   // y := R2
```

## Access to non-local variables

```
void p1() {
  int x = 1;
  int y = 2;
  void p2() {
    int z = 3;
    void p3() {
      int t = 6;
PC->  y = z;
    }
    p3();
   }
  p2();
}
```






## Non-local access

What are the addresses of y and z?
Follow the static link a number of steps to get to the appropriate activation. Then use local access relative to that activation.
The address of z:

$$\begin{aligned} FPp2 &= M[FP + offset(statlink)] = M[FP + 2] \\ zAddr &= FPp2 + hSize + offset(p2.z) = FPp2 + 3 + 0 \end{aligned}$$

which can be simplified to $zAddr = M[FP + 2] + 3$

The address of y:

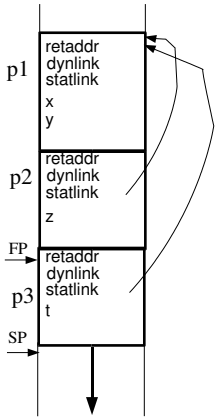
$$\begin{aligned} FPp2 &= M[FP + offset(statlink)] \\ FPp1 &= M[FPp2 + offset(statlink)] \\ yAddr &= FPp1 + hSize + offset(p1.y) \end{aligned}$$

which can be simplified to $yAddr = M[M[FP + 2] + 2] + 4$



## Access to non-local variables

```
void p1() {
  int x = 1;
  int y = 2;
  void p2() {
    int z = 3;
    p3();
   }
  void p3() {
    int t = 6;
PC->y = t;
  }
  p2();
}
```






## Non-local access

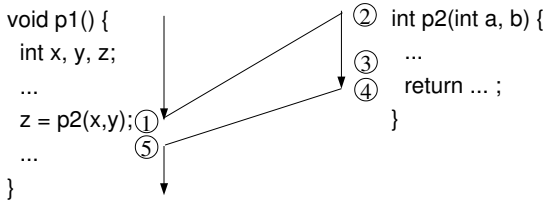
What is the addresses of y?

$$\begin{aligned} FPp1 &= M[FP + offset(statlink)] \\ yAddr &= FPp1 + hSize + offset(p1.y) \end{aligned}$$

which can be simplified to $yAddr = M[FP + 2] + 4$

If the frame pointer points to the static link rather than the beginning of the frame the formulas are somewhat simpler.

## Method call

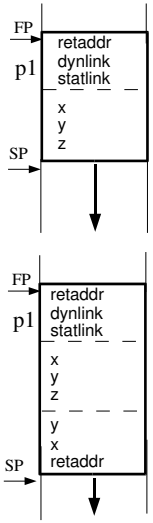



1. Compute and transfer arguments and return address. Jump to the code of the new activation (p2).
2. Allocate the new activation, set dynlink and statlink, update FP and SP.
3. Execute the code for p2
4. Transfer the return value. Deallocate the current activation (restore FP and SP). Jump back.
5. Deallocate arguments, save the return value (if needed), and continue execution.



## Step 1, Intel style

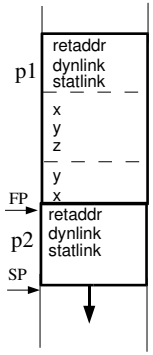
1. Compute arguments and push the values in reverse order on the stack.
2. Compute the new static link and transfer it to the called procedure using a register.
3. Execute a call instruction. The return address will be pushed on the stack.






## Step 2

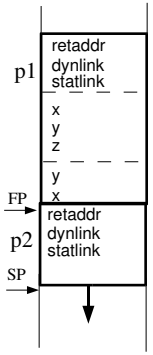
1. Push the current frame pointer (dynamic link).
2. Compute the static link and push it.
3. Set the frame pointer
4. Allocate memory for local variables etc.





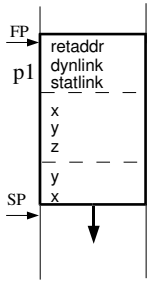

## Step 3

1. Execute the code for p2



## Step 4

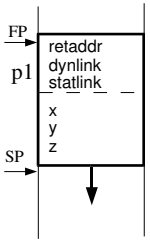
1. Transfer the return value
2. Deallocate the current activation (restore FP and SP)
3. Jump back to the return address






## Step 5

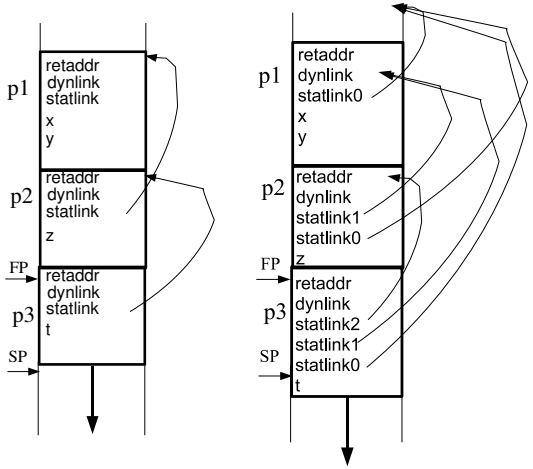
1. Save the return value
2. Deallocate arguments






## Access to non-local variables using a display

```
void p1() {
  int x = 1;
  int y = 2;
  void p2() {
    int z = 3;
    void p3() {
      int t = 6;
PC->  y = z;
    }
    p3();
  }
  p2();
}
```






## Data needed to generate code

For each variable and argument declaration

- local offset

For each use of a variable or argument

- the number of block levels (static levels) to the declaration (in order to find the appropriate activation by following static links)

For each method call

- the number of block levels (static levels) to the declaration of the method (in order to be able to set its static link).

For each method declaration

- the space needed for local declarations and temporary variables (in order to allocate an activation record of appropriate size)

## Typical optimizations for method calls

Store data in registers instead of in the activation record:

- return value
- the n first arguments
- the static and dynamic links (viewed as special arguments)
- the return address
  Usually, there is a special jump instruction that automatically stores the next address (the return address)

The called method must save these values in its activation record if it calls other methods.



## Calling conventions

Conventions for which activation has the responsibility for saving register values

**Caller-save register**: A method must save these registers before calling another method.

**Callee-save register**: The called method must save these registers before using them, and restore them before return.



## Many variations on activation layout

- The stack is not upside down
- SP point one word off (compared to our layout)
- FP points to the static link
- All static links are copied into the activation record
- The arguments for an activation are located in the current frame.
- The static link is viewed as a special argument.
- No dynamic link (the frame size is stored directly in the code instead)
- The frame is allocated in one step (instead of pushing)
- A bounded stack of register frames is used for the arguments



## Object-oriented languages

1. Objects — are allocated on a *heap*
   This is not a "priority queue heap".
   A stack cannot be used.
2. Inheritance
3. Method calls
4. Overriding and dynamic dispatch by means of a "v-table" (virtual table)

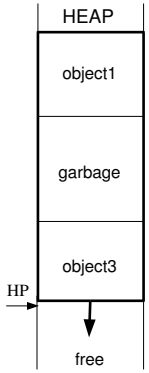
## Heap of objects

An area in memory is reserved for the heap

- each class instance is represented by an object for storing instance variables, etc.
- when an object is no longer needed, the corresponding memory can be reused for another object. (see F13 about automatic memory management)

Dedicated registers

- HP *Heap Pointer* points to the first free word in the heap





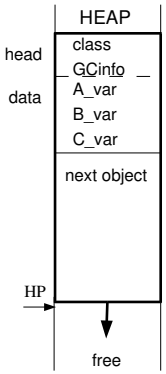

## Example of object layout

Head

- class — class pointer — points to a description of the class (with, e.g., a virtual table)
- GCinfo — Garbage Collection info — usually, a word is needed by the GC (this depends on the algorithm used)

Data

- X_var — instance variable from class X
- Prefixing: Inherited instance variables are located first in the object (Suppose C is a subclass of B, and B a subclass of A).
- Block structure: If the class is located inside another block, the object can have one (or more) static links (viewed as special instance variables).






## Example of layout for class description

Head

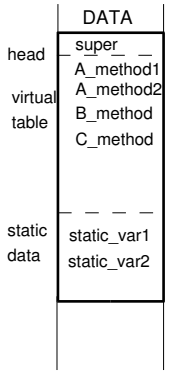
- super — pointer to the description of the superclass

Virtual table

- X_method$_j$ — address to the code for method m declared in class X
- Prefixing: (Addresses to) methods inherited from superclasses are located first in the description object. (Suppose C is a subclass of B, and B a subclass of A.)

Static variables

- static_var$_k$ — static variable declared in the class



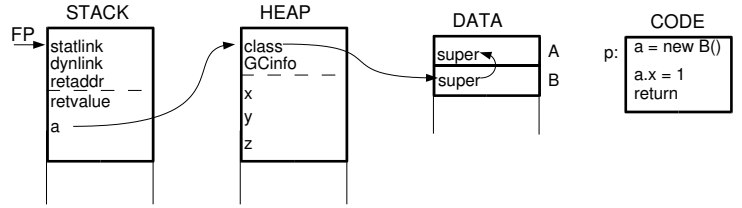



## Inheritance of instance variables

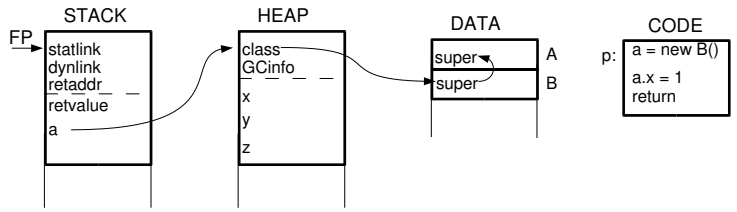
```
void p() {
  A a = new B()
  a.x = 1;
}
```

```
class A {
  int x;
  int y;
}
```

```
class B extends A {
  int z;
}
```



## Corresponding execution




The statement a.x = 1

$$aObj = M[FP + hSize + offset(a)]$$
$$M[aObj + ohSize + offset(x)] = 1$$

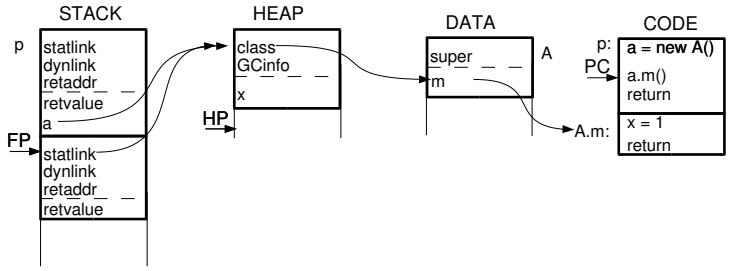
where ohSize is the object header size or, simplified
$M[M[FP + 4] + 2] = 1$
The access to a.x is independent of the dynamic type of a (A or B).



## Method call

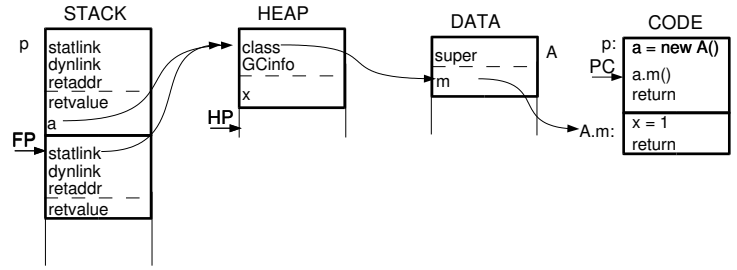
```
void p() {
  A a = new A()
  a.m();
}
```

```
class A {
  int x;
    void m() {
      x = 1;
    }
}
```






## Corresponding execution




The static link points to the enclosing object.
NB! The "this" pointer in a method is the same as the static link!



## Create an object

| statement | step | what happens |
|---|---|---|
| a = new A() | allocate a new A object | objAddr = HP<br>HP = HP + sizeOf(A) |
| | set the class pointer | M[objAddr] = A |
| | set the GCinfo | M[objAddr+1] = ... |
| | initialize variables | ... |
| | run the constructor | ... |
| | assign the variable a | M[FP+4] = objAddr |

## Call method

| statement | step | what happens |
|---|---|---|
| a.m() | set static link | M[SP] = M[FP+hSize+offset(p.a)] |
| | set dynamic link | M[SP+1] = FP |
| | set the return address | M[SP+2] = PC+2 |
| | compute the address of the method a.m via the v-table | aAddr = M[FP+hSize+offset(p.a)]<br>clsAddr = M[aAddr]<br>mAddr = M[clsAddr+chSize+offset(A.m)]<br>where chSize is the size of a class header. Or, simplified<br>mAddr = M[M[M[FP+3]]+1] |
| | jump to the code of a.m | Jump mAddr |

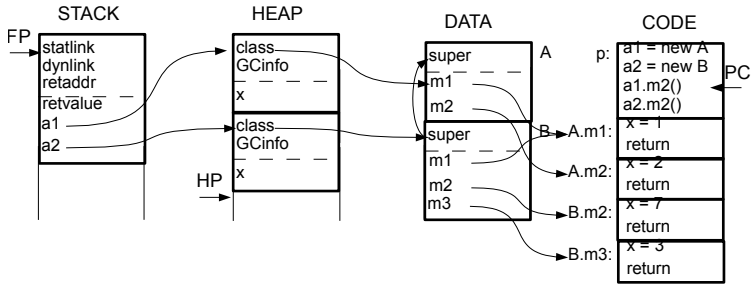


## Overriding and dynamic dispatch

```
void p() {
  A a1 = new A();
  A a2 = new B();
  a1.m2();
  a2.m2();
}
class A {
  int x;
  void m1() { x = 1; }
  void m2() { x = 2; }
}
class B extends A {
  void m2() { x = 7; }
  void m3() { x = 3; }
}
```



## Corresponding execution



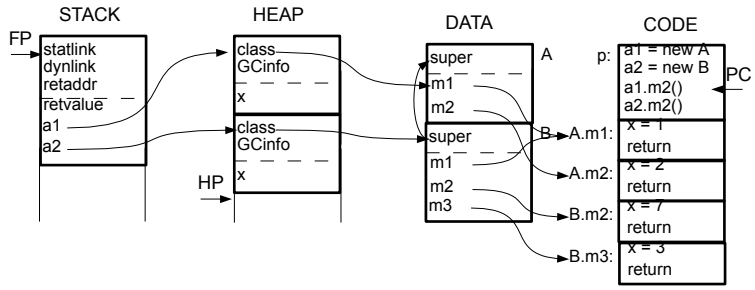

```
void p() {
  A a1 = new A();
  A a2 = new B();
  a1.m2();
  a2.m2();
}
```

```
class A {
  int x;
  void m1()
  void m2()
}
```

```
class B extends A {
  void m2() { }
  void m3() { }
}
```



## Corresponding execution




The method m2 is at the same offset in the virtual tables of both A and B.
The code for calls to obj.m2 is thus independent of the dynamic type of obj.