(Greatness from ...) Small beginnings

\[ \text{plus}_1 :: (\text{Num} \; a) \rightarrow a \]
\[ \text{plus}_1 \; x = x + 1 \]

```c
int plus1 (int x) {
    return x + 1;
}
```

```python
def plus1 (x):
    return x + 1;
```

```fortran
pure function plus_1 (x)
int, intent (in) :: x
int :: plus_1
plus_1 = x + 1;
end function;
```

```pascal```
function Plus_1 (x : integer) : integer;
begin  Plus_1 := x + 1; end;
```

References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 “Instructions: Language of the Computer”
ARM edition, Morgan Kaufmann 2017
156

Functions

mov r0, #1
bl Plus_1
mov r4, r0

Plus_1:
add r0, #1
bx lr

157

Functions

lr

mov r0, #1
bl Plus_1
mov r4, r0

Plus_1:
add r0, #1
bx lr

158

Functions

mov r0, #1
bl Plus_1
mov r4, r0

Plus_1:
add r0, #1
bx lr

159

Functions

Can/should this always be done?

This is called inlining

Can/should this always be done?
Functions

(Greatness from ...) Small beginnings

```haskell
plus_2 :: (Num a) => a -> a
plus_2 x = plus_1 $ plus_1 x

int plus2 (int x) {
    return plus1 (plus1 (x));
}
```

```haskell
function Plus_2 (x : Integer) return Integer is (Plus_1 (Plus_1 (x)));
```

```haskell
plus_2 = plus_1 (plus_1 (x));
```

```haskell
pure function plus_2 (x)
int, intent (in) :: x
int :: plus_2
plus_2 = plus_1 (plus_1 (x));
end function;
```

```haskell
function Plus_2 (x : integer) : integer;
begin  Plus_2 := Plus_1 (Plus_1 (x)); end;
```

```
(bx lr)
```

```
What is the value of lr in each case?
```
... we need an example, where a compiler will not just remove all our code!

... where does this lead us?
What happens to our parameter \( x \) during the function? What if this was holding some information at the time when we were called?
Keeping a reference to the start of the Stack Frame for this function (with the frame-pointer \(fp\)) makes things neater and enables structured access to the dynamic context.

While addressing via the \(sp\) is possible, it may also be complex to keep track of, as the \(sp\) may change further.
Functions

Is Fact reentrant?

Fact:

unsigned int fact (unsigned int x) {  
if (x == 0) return 1;  
else return x * fact (x - 1);  
}

A compiler will likely replace such a recursion!

unsigned int fact (unsigned int x) {  
    fac = 1;  
    for (i = 1, i <= x, i++) {  
        fac = fac * i;  
    }  
    return fac;  
}

A compiler will likely replace such a recursion!

 besides all the inlining, unrolling, flattening, etc.
= Stack operations are still vital for any non-trivial program.

© 2021 Uwe R. Zimmer, The Australian National University
Functions

Fib:
... ...

Fib_Fact:
...

Fact:
... ...

Fact_Loop:
... ...

End_Fact:
... ...

We keep a copy of G here.

But we don’t save r3?

But we don’t keep a copy of r0 here!

What would be the maximal depth of the stack?

What would the stack look like?

We save a copy of r4 here.

But we don’t save r3?

There could be two further Fib-calls for each call to Fib...

What would be the maximal depth for the stack?

There could be two further Fib-calls for each call to Fib...

We save a copy of r4 here.

What would the stack look like?
Functions

Components / phases of a function call:

- Values (parameters) to be passed to a function.
- Local variables inside a function.
- Values (results) to be returned from a function.

So far we:
- ... passed parameter values in registers (r0 - r3).
- ... called the function (store the return address and jump to the beginning of the function).
- ... pushed the return address, the previous stack frame and used registers (r4 ...).
- ... created a new stack frame (and addressed all local variables relative to this).
- ... grew the stack such that it can hold the local variables.
- ... done the calculations/operations based on the local variables and scratch registers.

If those registers do not suffice, additional parameters and results are passed via the stack.

- There are also memory alignment constraints. (Mostly due to memory bus constraints)
- Why are these conventions architecture related at all?

© 2021 Uwe R. Zimmer, The Australian National University page 184 of 487 (chapter 3: "Functions" up to page 232)

185

Functions

Components / phases of a function call:

- Values (parameters) to be passed to a function.
- Local variables inside a function.
- Values (results) to be returned from a function.

So far we:
- ... passed parameter values in registers (r0 - r3).
- ... called the function (store the return address and jump to the beginning of the function).
- ... pushed the return address, the previous stack frame and used registers (r4 ...).
- ... created a new stack frame (and addressed all local variables relative to this).
- ... grew the stack such that it can hold the local variables.
- ... done the calculations/operations based on the local variables and scratch registers.
- ... passed return values in registers (r0 - r1).
- ... restored the stack pointer (and thus de-allocated all local variables).
- ... popped the return address, the previous stack frame and used registers (r4 ...).
- ... jumped back to the next instruction after the original function call.
- ... used the return values found in r0 - r1.

© 2021 Uwe R. Zimmer, The Australian National University page 185 of 487 (chapter 3: "Functions" up to page 232)

186

Functions

Conventions

ARM architecture calling practice

- r0-r3 are used for parameters.
- r0-r1 are used for return values.
- r0-r3 are not expected to be intact after a function call ... all other registers are expected to be intact!
- If those registers do not suffice, additional parameters and results are passed via the stack.

- Conventions are different in other architectures (e.g. x86, where parameters are generally passed via the stack).
- There are also memory alignment constraints. (Mostly due to memory bus constraints)
- Why are these conventions architecture related at all?

© 2021 Uwe R. Zimmer, The Australian National University page 186 of 487 (chapter 3: "Functions" up to page 232)

187

Functions

Parameter passing

Call by ...

<table>
<thead>
<tr>
<th>Information flow</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>by reference (mutable, no read)</td>
</tr>
<tr>
<td>by result</td>
<td>No read access from inside the function, write access on return.</td>
<td></td>
</tr>
<tr>
<td>in &amp; out</td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td></td>
</tr>
<tr>
<td>by value result</td>
<td>Function can read and write at any time. Outside code shall not write.</td>
<td></td>
</tr>
</tbody>
</table>

© 2021 Uwe R. Zimmer, The Australian National University page 187 of 487 (chapter 3: "Functions" up to page 232)
**Haskell**

Only control over information flow – not over access.

<table>
<thead>
<tr>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>in &amp; out</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
</tbody>
</table>

**Python**

All parameter access is double-indirect (i.e., handles).

<table>
<thead>
<tr>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>in &amp; out</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
</tbody>
</table>

**Ada**

Limited control over "by value result". "by value" parameters are constants.

<table>
<thead>
<tr>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>in &amp; out</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
</tr>
<tr>
<td></td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
</tbody>
</table>
**Functions**

**Assembly**

"By reference" semantics by convention only.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| by result        | by reference (mutable, no read) |         |             |
| by value result  | Function can read and write at any time. Outside code shall not write. |         |             |

**Parameter passing**

**Call by name**

... is conceptually a call-by-value, where the value has not been calculated yet.

Technically a reference to a function is passed and the evaluation of this parameter (function) is left to the called function.

Features:

- Values are only evaluated if and when they are needed.
- Values can change during the life-time of a function (in case of side-effecting functions).
- Values can be stored once calculated (in case of side-effect-free functions).

While this is possible to a degree in most programming languages ...

... it is a core concept for functional, lazy evaluation languages, like e.g. Haskell, and it does find its way back into mainstream languages like C++, .NET languages or Python as anonymous functions (sometimes referred to as λ-functions or λ-expressions).
Fact 2

No read access from inside the function or is copied into a local variable.

Fact 3

Parameter becomes a constant while the function runs (also from outside the function).

Parameter passing

Call by ...

by copy

Parameter becomes a constant inside the function or is copied into a local variable.

by reference

Parameter becomes a constant while the function runs (also from outside the function).

We should have used either of those modes.

by result

Calling function expects the parameter value to appear in a specific space at return.

by value result

Function can read and write at any time. Outside code shall not write.

Yet we used this mode.

What is the value of x during one execution of Fact?

Case_Others:

sub r0, #1
add r0, r0, #4
add sp, #4
ldr sp, (r4, fp, lr)

Fib_Fact:

mov r1, r0
sub sp, #4

End_Fact:

mov sp, fp
ldmia sp, (r5, fp, lr)
bx lr
### One-way and by-copy

Those are side-effect-free and hence the resulting scenarios are easy to analyse.

**Information flow**

- **by copy**
  - in: Parameter becomes a constant inside the function or is copied into a local variable.
  - out: Calling function expects the parameter value to appear in a specific space at return.
  - in & out: Parameter is copied to a local variable and copied back at return.

- **by reference**
  - in: No write access is allowed while the function runs (also from outside the function).
  - out: No read access from inside the function, write access on return.
  - in & out: Function can read and write at any time. Outside code shall not write.

### Two-way and by-copy

Still side-effect-free within the function (but not on the outside). Potentially more convenient as memory space can be reused.

**Information flow**

- **by copy**
  - in: Parameter becomes a constant inside the function or is copied into a local variable.
  - out: Calling function expects the parameter value to appear in a specific space at return.
  - in & out: Parameter is copied to a local variable and copied back at return.

- **by reference**
  - in: No write access is allowed while the function runs (also from outside the function).
  - out: No read access from inside the function, write access on return.
  - in & out: Function can read and write at any time. Outside code shall not write.

### Two-way and by-reference

Side-effecting and particular care is required as multiple entities could write on this.

No data has to be replicated.

**Information flow**

- **by copy**
  - in: Parameter becomes a constant inside the function or is copied into a local variable.
  - out: Calling function expects the parameter value to appear in a specific space at return.
  - in & out: Parameter is copied to a local variable and copied back at return.

- **by reference**
  - in: No write access is allowed while the function runs (also from outside the function).
  - out: No read access from inside the function, write access on return.
  - in & out: Function can read and write at any time. Outside code shall not write.

### One-way-Out and by-reference

Side-effect-free, if new memory is allocated on return – cannot be enforced on assembly level (requires compiler).

**Information flow**

- **by copy**
  - in: Parameter becomes a constant inside the function or is copied into a local variable.
  - out: Calling function expects the parameter value to appear in a specific space at return.
  - in & out: Parameter is copied to a local variable and copied back at return.

- **by reference**
  - in: No write access is allowed while the function runs (also from outside the function).
  - out: No read access from inside the function, write access on return.
  - in & out: Function can read and write at any time. Outside code shall not write.
**Generic Stack-Frame**

The current code prepared to call a function:

- Push parameters on the stack.
- Works for any data size (unless the stack overflows)
- and parameter passing mode.

---

**Functions**

- Types, storage structures and passing modes must be agreed upon between caller and callee.
- Solved if the languages, compiler and program are the same.
- If the languages or compilers are different, standards will be required.
Functions

Generic Stack-Frame

Functions (in programming languages) have a context.
⇒ E.g., the surrounding function or the hosting object.
The caller knows this context and provides it.

The caller also provides a reference to its own stack frame.
⇒ This builds a linear chain of calls through the stack.
Will be used e.g., for debugging (stack trace)
and exception propagation.

Some languages will not have a context by default, like C or Assembly.
(gnu C expands the C standard and provides it though)

Generic Stack-Frame

The caller also provides a reference to its own stack frame.
⇒ This builds a linear chain of calls through the stack.
Will be used e.g., for debugging (stack trace)
and exception propagation.

⇒ The static and dynamic link might be identical in some cases.

Static vs. dynamic links

function a (x : Integer) return Integer is
    function b (y : Integer) return Integer is (x + y);
    function c (z : Integer) return Integer is (b (z));
begin
    return c (x);
end a;

a :: Integer -> Integer
a x = c x

where
    b :: Integer -> Integer
    b y = x + y
    c :: Integer -> Integer
    c z = b z
Functions

Static vs. dynamic links

function a (x : Integer) return Integer is
    function b (y : Integer) return Integer is (x + y);
    function c (z : Integer) return Integer is (b (z));
    begin
        return c (x);
    end a;

a :: Integer -> Integer
a x = c x
where
    b :: Integer -> Integer
    b y = x + y
    c :: Integer -> Integer
    c z = b z

⇒ The dynamic and static link for function c are both function a.

⇒ The caller of function b is function c.
⇒ Hence exceptions raised in b are handled first in b, then in c, and then a.

⇒ The dynamic and static link for function c are both function a.

⇒ The context for function b is function a.
⇒ Hence b can access x (but not z).

Dynamic link (prior frame)
⇒ The caller of function b is function c.
⇒ Hence exceptions raised in b are handled first in b, then in c, and then a.
⇒ The dynamic and static link for function c are both function a.

⇒ The dynamic and static link for function c are both function a.

Generic Stack-Frame

The caller also provides a reference to its own stack frame.
⇒ This builds a linear chain of calls through the stack.
Will be used e.g. for debugging (stack trace) and exception propagation.
Generic Stack-Frame

A Frame Pointer (FP) is established at the boundary between the caller and the callee. Operations from here are in the control of the callee.

The last item to be stored before handing over control is the address of the following instruction. The control flow can later return to this address.

The Instruction Pointer (IP), sometimes also called Program Counter (PC), is changed to a new address. Operations from here are in the control of the callee.

Local variables are allocated (by moving the stack pointer). Local variables can be of any size or structure, unless the stack overflows.

Operations from here are in the control of the callee.
Functions

Generic Stack-Frame

Local variables are allocated (by moving the stack pointer). Local variables can be of any size or structure, unless the stack overflows.

- The completes a new stack frame.
- Variables and parameters from the context stay visible (via the chain of static links).

While this function is executing, local variables can still be added.

Handy, if e.g. the size of a local variable is not yet determined when the function starts.

The next function call will produce the next stack frame.

- Variables and parameters from the context stay visible (via the chain of static links).

Local variables can only be added to the currently executing function.
Functions

**Generic Stack-Frame**

The next function call will produce the next stack frame.

- Note which variables and parameters are visible.

Accessing the context like that can be inefficient!

Compilers may choose other mechanism e.g., displays, which make all context levels accessible at once.

**Pre_Call**:
- Allocate/identify space for the parameters
- Copy the in and in-out parameters to this space
- Potentially provide links
- Provide a return address ("Post_Call")
- (usually implicit with the call itself)

**Call the function**

**Post_Call**:
- Copy the out and in-out parameters
- to local variables or registers
- Potentially restore the frame pointer
- Restore the stack to its previous state
- (if the stack has been used)

**Return from function**
Functions

Generic Stack-Frame – Heap

How to keep any memory allocation after function return?

By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

... and a local variable in the calling function can keep this link.

When to deallocate though?

• Garbage collection (Java)?
• Smart pointers (C++)?
• Scoped pointers / storage pools (Ada)?
• Reference ownerships (Rust)?
• Scopped pointers / storage pools (Ada)?
Summary

Functions

• Framework
  • Return address
  • Relative addressing

• Parameter passing modes and mechanisms
  • Copy versus reference
  • Information flow directions
  • Late evaluation

• Stackframes
  • Static and dynamic links
  • Parameters
  • Local variables