# Code Generation for VSOP

Cyril Soldani, Florent De Geeter

INFO0085

April 26, 2022

# Outline

# Code Generation


For semantically valid VSOP programs, generate LLVM IR.

Use generated IR to build native executable.

# The assignment

Due at the latest for the **15th of May**.

`vsopc /path/example.vsop` will generate `/path/example` executable, which will then be run without argument. If `/path/example.input` exists, it will be used as `stdin`.

`vsopc -i example.vsop` will dump the LLVM IR on `stdout`.

Your compiler should not output any error at code generation, **all errors should already have been reported**. Of course, it will still report potential lexical, syntax or semantic errors, and an assertion failure is better than a segfault or generating invalid code.

You can implement some **VSOP extensions** of your liking. Keep the basic VSOP compiler, extensions should be recognized **only** with the `-e` argument. `-e -i` should be supported.

Provide tests in `tests` sub-folder.

# This part is optional

This part is not mandatory, but not doing it will **limit the maximal grade** you can get for this course to **13 out of 20**.

If you decide to not do this part, you still have to **submit your code** and **a report** on the submission platform.

You thus still have the opportunity to improve your codes of the 3 first parts.

# Semantics of VSOP is given in the manual

- Scoping rules.
- Typing rules.
- **Evaluation rules**.

The code generated **at compile-time** should follow the evaluation semantics **at run-time**.

# VSOP has dynamic dispatch (reminder)

The method which is called depends on the **dynamic type** of an object, not its **static type**.

```
class P { name() : string { "P" } }
class C extends P {
  name() : string { "C" }
  onlyInC() : int32 { (* ... *) } }
class Other {
  myMethod() : string {
    let p : P <-            // Declared type is P => static type is P.
      if inputInt32() = 0   // inputInt32() will ask the user for a number.
        then new C          // `new C` valid here as C conforms to P.
        else new P
    in {
      p.onlyInC(); // Type error. Static type is P, not C. Would be valid
                   // if the user typed 0, but we cannot tell at compile
                   // time.
      p.name() // Dispatch is done using dynamic type.
               // Will return "P" or "C" depending on what the user typed.
    } } }
```

# Report

Provide a PDF report describing:

- How your code is organised, which tools, data structures and algorithms are used.
- The potential shift/reduce or reduce/reduce conflicts your parser may have.
- If something in your implementation is not obvious from your **documented** code.
- Your VSOP extensions.
- The **limitations** of your compiler. What would you do differently or with additional time?
- How long did it take, what were the main difficulties? How can we improve the project?

Try to be succinct

# Outline

# What is LLVM?

- A *Low-Level Virtual Machine*?

# What is LLVM?

- A *Low-Level Virtual Machine*? Not really.
- A collection of *modular and reusable* tools to support both *static and dynamic* compilation of arbitrary programming languages.

# What is LLVM?

- A *Low-Level Virtual Machine*? Not really.
- A collection of *modular and reusable* tools to support both *static and dynamic* compilation of arbitrary programming languages.
- Industrial-strength tools and library.
- Many sub-projects, the most prominent one being *clang*, a C/C++ compiler competitive with GCC (and with arguably better error messages).
- Wildly used in the industry (e.g. Adobe, Apple, Cray, Intel, NVIDIA, Siemens, Sun, ...).
- Used by the cool kids on the block: Haskell, rust, julia, *etc*.
- Used in high-performance computing for data specialization.
- Somewhat *easy*, *fast*, *modular*, and supports many targets.

# What is the LLVM Language?

- An *Intermediate Representation* (IR)
- Assembly-like language (remember beta assembly?)
- Based on the *Single Static Assignment* (SSA) form
- Typed!
- Allows low-level operations ...
- ...but permits to represent high-level languages cleanly (*e.g.* allows function definitions)
- Either textual, or bitcode
- All LLVM tools (optimization passes, native compilers) acts on this representation

# Resources

- LLVM Language Reference Manual
  https://releases.llvm.org/11.0.0/docs/LangRef.html
- The Often Misunderstood GEP Instruction
  https://releases.llvm.org/11.0.0/docs/GetElementPtr.html
- **The LLVM Tutorial**
  Kaleidoscope: Implementing a Language with LLVM
  https://releases.llvm.org/11.0.0/docs/tutorial/index.html
- Static Single Assignment (SSA) Form (Wikipedia)
  http://en.wikipedia.org/wiki/Static_single_assignment_form
- The (not so) theoretical course!
- Mapping High-Level Constructs to LLVM IR
  https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/README.html

# Practical Details

## Version

Use LLVM version 11 (or 9) available in the reference container.

## How is LLVM Code Generated?

- **Textually**, then call LLVM assembler (and optimizer)
- Using the LLVM library:
  - Easier to optimize
  - Allows interpretation and JIT compilation
  - **Poorly documented**, steep learning curve!

## Implementation Language

- Written in C++
- Official bindings for C and OCaml
- Various unofficial bindings in other languages (beware of their quality!)

# Modules

- A *compilation unit* is called a **module**
- A module contains:
  - type aliases
  - declarations, which states that a symbol exists and give its type
  - definitions of global variables (type + initializer)
  - function definitions
  - special sections such as module initialization code
  - all of the above in the *LLVM Language*
- A module lies in its own file (`module_name.ll`)

  ... or is generated in-memory using the library
- You are expected to generate a single module in this project

# Types

```
i1                    ; 1-bit integer (may be used for bool)
i8                    ; 8-bit integer (used for char)
i32                   ; 32-bit integer
i64                   ; 64-bit integer
float                 ; 32-bit IEEE 754 floating-point number
double                ; 64-bit IEEE 754 floating-point number
void                  ; empty type
label                 ; named adresses (see later)
i32 *                 ; pointer to one (or more) i32
[40 x i8]             ; array of 40 8-bit numbers (static size)
float (i16, i32 *)    ; function (i16, i32 *) -> float
i32 (i8 *, ...)       ; signature of printf()
{ i32, float }        ; structure with i32 and float fields
%MyVeryOwnType        ; named type
```

# SSA Values

- Also known as, *registers* (somewhat improperly)
- They are **immutable**, *i.e.* do not change after initialization

### Two kinds of identifiers

- Global identifiers (functions, global variables) begin with the `@` character (*e.g.* `@puts`, `@my_global_var`)
- Local identifiers (register names) and types begin with the % character (*e.g.* `%result`, `%MyStruct`)

- Infinite supply of registers and globals
- Unnamed values are (and **must** be) numbered sequentially (*e.g.* `@404`, %42)

# Literals

## Simple

- Integers: *e.g.* 42, -1028
- Booleans: `true` and `false` of `i1` type (resp. 1 and 0)
- Floating-point: usual notation (*e.g.* 3.14159265, 6.67398e-11), but requires **exact decimal value** (*e.g.* 0.1 is rejected), or direct representation (*e.g.* `0x141d7038`)
- Null pointer: `null`, of any pointer type

## Complex

- Structures: *e.g.* `{ i32 4, float 17.0, i32* @g }`
- Arrays: `[i32 1, i32 2, i32 3, i32 4, i32 5]`
- Zeroes: `zeroinitializer`
- Undefined: `undef`
- Global variables and functions are always implicit pointers (*i.e.* `@x` always has a pointer type)

# Blocks, Labels and Terminator Instructions

A function body is a set of SSA **blocks**.
A block spans from a **label** to a **terminating instruction**.

```
ret void                   ; return from procedure
ret i32 42                 ; return 42
ret { i32, double } { i32 42, double 84.0 } ; return struct literal
br label %end_of_process   ; unconditional jump
br i1 %cond, label %if_true, label %if_false  ; conditional jump
```

Example:

```
    br i1 %cond, label %if_true, label %if_false
    ; A statement here would be illegal
if_true:            ; Labels are as in C
    ret i32 42
if_false:
    ret i32 0
```

# Arithmetic Instructions

- Separate instructions for integers and floating-point numbers
- Operands type (and size) must be known
- No signedness in types, but signedness in some instructions

```
%1 = add i32 42, %var          ; %var must also be i32
%2 = add i8 64, %c             ; type can be any integer
%3 = fadd double 42.0, %x      ; different instruction for floats
%4 = sub i32 0, %var           ; yields -%var
%5 = fmul double 3.0, %x       ; yields 3.0 * %x
%6 = udiv i32 %var, 5          ; unsigned division
%7 = sdiv i32 %var, 5          ; signed division (rounded towards 0)
%8 = fdiv double %x, 5.0
```

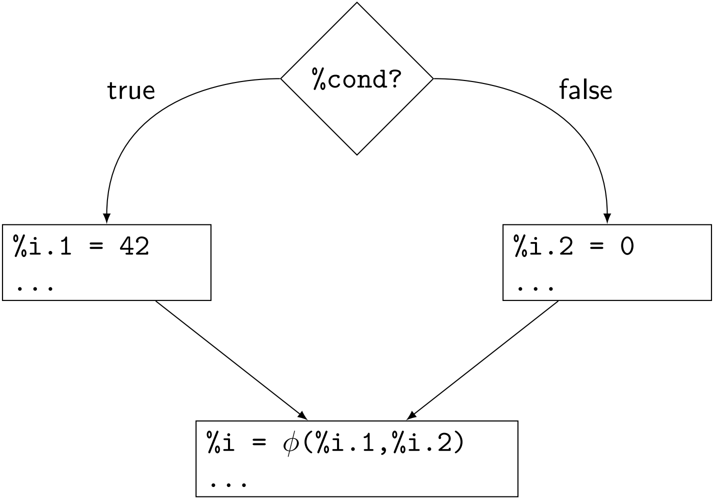
There are other operations (and flags), see full reference.

# Logical Instructions and Comparisons

```
%1 = and i1 %a, %b
%2 = and i32 9, 10  ; bitwise, yields 8 (1001 & 1010 = 1000)
%3 = or i32 9, 10   ; yields 11 (1001 | 1010 = 1011)
%4 = xor i32 9, 10  ; yields 3 (1001 ^ 1010 = 0011)
%5 = xor i32 %x, -1 ; yields ~%x
```

```
%1 = icmp eq i32 %a, %b       ; %a == %b, returns i1 type
%2 = icmp ne i32 %a, %b       ; %a != %b
%3 = icmp ugt i32 %a, %b      ; unsigned %a > %b
%4 = icmp sle i32 %a, %b      ; signed %a <= %b
%5 = icmp eq i32* @p, @q      ; works also on pointers
%7 = fcmp oeq float %x, %y    ; ordered equal, i.e.
                              ; %x != NaN && %y != NaN && %x == %y
%8 = fcmp ueq float %x, %y    ; unordered equal, i.e.
                              ; %x == NaN || %y == NaN || %x == %y
%9 = fcmp olt float %x, %y    ; %x != NaN && %y != NaN && %x < %y
```

# The Magical Phi Instruction: Principle



# The Magical Phi Instruction: in LLVM

```
  br i1 %cond, label %if_true, label %if_false
if_true:
  %i.1 = i32 42
  br label %end_if
if_false:
  %i.2 = i32 0
  br label %end_if
end_if:
  %i = phi i32 [%i.1, %if_true], [%i.2, %if_false]
```

- `phi` takes on the value specified by the pair corresponding to the block that executed **just before** the current block
- constant values are allowed as well

# Functions Declarations and Calls

Declarations

```
declare i32 @printf(i8*, ...)
declare ccc i32 @printf(i8*, ...)
declare fastcc void @someFastCCFun(i32, i8*)
```

Calls

```
%z = call double @pow(double %x, double %y)
call i32(i8*, ...)* @printf(i8* @fmt, i32 %val)
call fastcc void @someFastCCFun(i32 42, i8* @str)
```

- Functions are **always pointers**
- Catching the return value is not mandatory
- Catching void is **illegal**!
- Type can be just the return type, unless vararg

# Memory Accesses

```
%ptr = alloca i32             ; yields i32* to STACK-allocated i32
store i32 42, i32* %ptr       ; stores 42 into allocated cell
%val = load i32, i32* %ptr    ; %val := 42 (read from cell)

%aPtr = alloca i32, i64 100 ; int32_t aPtr[100];
; fill(aPtr, 100)
call fastcc void @fill(i32* %aPtr, i32 100)
; int *a4ptr = &(aPtr[4]);
%a4ptr = getelementptr i32, i32* %aPtr, i64 4
%a4 = load i32, i32* %a4ptr ; int a4 = *a4ptr;

%sPtr = alloca {i32, i8}      ; struct { int32_t a, int8_t b } s;
%bPtr = getelementptr {i32, i8}, {i32, i8}* %sPtr, i32 0, i32 1
store i8 42, i8* %bPtr        ; s.b = 42;
```

- **No memory dereference** in `getelementptr`
- Structures are indexed with `i32`, any integer for arrays
- Use `malloc` for heap-allocated memory

# Memory Accesses

```
%ptr = alloca i32              ; yields i32* to STACK-allocated i32
store i32 42, i32* %ptr        ; stores 42 into allocated cell
%val = load i32, i32* %ptr     ; %val := 42 (read from cell)

%aPtr = alloca i32, i64 100 ; int32_t aPtr[100];
; fill(aPtr, 100)
call fastcc void @fill(i32* %aPtr, i32 100)
; int *a4ptr = &(aPtr[4]);
%a4ptr = getelementptr i32, i32* %aPtr, i64 4
%a4 = load i32, i32* %a4ptr ; int a4 = *a4ptr;

%sPtr = alloca {i32, i8}       ; struct { int32_t a, int8_t b } s;
%bPtr = getelementptr {i32, i8}, {i32, i8}* %sPtr, i32 0, i32 1
store i8 42, i8* %bPtr         ; s.b = 42;
```

- **No memory dereference** in `getelementptr`
- Structures are indexed with `i32`, any integer for arrays
- Use `malloc` for heap-allocated memory
- First index is **always** for the pointer type

# Function Definitions

```
define fastcc void @fill(i32* %array, i32 %len) {
entry:  ; If not specified, label %0 will be inserted
    br label %loop_cond

loop_cond:
    %i = phi i32 [ 0, %entry], [ %ip1, %loop_body ]
    %cond = icmp ult i32 %i, %len
    br i1 %cond, label %loop_body, label %loop_end

loop_body:
    %ptr = getelementptr i32, i32* %array, i32 %i
    store i32 %i, i32* %ptr
    %ip1 = add i32 %i, 1
    br label %loop_cond

loop_end:
    ret void
}
```

# Trick to Avoid Phi

- Store all variables in memory

```
  %iPtr = alloca i32
  br i1 %cond, label %if_true, label %if_false
if_true:
  store i32 42, i32* %iPtr
  br label %end_if
if_false:
  store i32 0, i32* %iPtr
  br label %end_if
end_if:
  %i = load i32, i32* %iPtr
```

- Then, optimize stack-allocated variables away with `mem2reg`
- Not worth it here, but may be in more complex code
- Approach used by `clang`

# Translation of lecture IR

Quite similar, but **typed**, and **no assignments**.

```
; LABEL my_label
my_label:
```

```
; GOTO my_label
br label %my_label
```

```
; id := 42     (7th assignment to id in that scope)
%id.6 = add i32 0, 42
; id := a + 1  (assumming i32 type)
%id = add i32 %a, 1
; id := g      (where g assigned in a if-then-else above)
%id = phi i32 [%g.1, %if_true], [%g.2, %if_false]
```

# Translation of lecture IR (cnt'd)

```
; id := M[addr]
%id = load i32, i32* %ptr
; M[addr] := id
store i32 %id, i32* %ptr
```

```
; a_exp_addr := exp * 4
; a_exp_addr := a_exp_addr + a_base_addr
%a_exp_ptr = getelementptr i32, i32* %a, i32 %exp
```

```
; IF n > 0 THEN if_greater ELSE if_lower
%cond = icmp sgt i32 %n, 0
br i1 %cond, label %if_greater, label %if_lower
```

```
; ret = CALL my_fun(arg1, arg2)
%val = call i32 (i32, i32)* @my_fun(i32 %arg1, i32 %arg2)
ret i32 %val
```

# How to tell the size of a struct?

Take the address of the second element of an *hypothetical* array of `%T` starting at address 0. As `getelementptr` returns a pointer, you then have to convert it back to an integer.

```
%size_as_ptr = getelementptr %T, %T* null, i32 1
%size_as_i32 = ptrtoint %T* %size_as_ptr to i32
```

You should avoid trying to compute the size yourself (like `clang` does), because of alignment issues which are target-dependent. *E.g.* on `x86_64`,

```
%T = type { i8, i8*, i32, i8, i32 }
```

will not have size 18 bytes, but 32 bytes. Apart from being target-dependent, the rules are not trivial. The type

```
%T2 = type { i8, i32, i8*, i8, i32 } ; 2nd <-> 3rd
```

only takes up 24 bytes.

# Strings

- All strings are ultimately string literals in basic VSOP
  - They are not mutable
  - They support no operation other than printing
- LLVM supports string literals natively, as arrays of i8

```
@s = constant [14 x i8] c"Hello, world!\00"

define i32 @main() {
    %1 = getelementptr [14 x i8], [14 x i8]* @s, i64 0, i64 0
    %2 = call i32 @puts(i8* %1)
    ret i32 0
}

declare i32 @puts(i8*)
```

- You can store them as i8*.

# What's Next

- We haven't covered all of LLVM language (not even close)
- Have a look at the language reference
- Have a look at the **tutorial**
- Use `clang -S -emit-llvm`
- Notable omissions:
  - tail call optimization
  - function and argument attributes
  - conversion and casting operations
  - LLVM intrinsics (*e.g.* for **pow**)
  - linkage types
  - undef and poison values
  - vectors and vectored operations
  - parallelism-oriented features
  - memory management features
  - special control flow (*e.g.* for exceptions)
  - low-level stuff (*e.g.* access to volatile memory)

# The LLVM library

The LLVM Library allows:

- declarations of (external) functions and global variables
- definitions of global variables and functions
- adding blocks to functions
- moving the insertion point (*builder position*) at start/end of any block
- emitting instructions
- checking functions/modules are well-formed
- executing optimization passes in any order

However, it is:

- poorly documented (see the tutorial and Doxygen);
- not mandatory.

A small example will be put on eCampus.

# Compilation/Execution with the Library

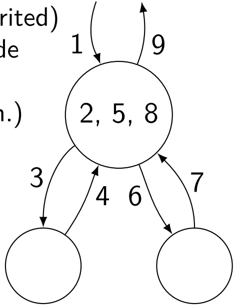
Once you built and optimized a module, you can:

- interpret it using a built-in interpreter
- execute it after *just-in-time* (JIT) compilation
- compile it into a native executable
- dump its LLVM assembly code (indeed, you can dump any LLVM value at any time)

# Outline

# AST Traversal Reminder

- You should refer to the (not so) theoretical course
- Generally a mix between top-down and bottom-up approaches
- At each node
    1. Receive some info from parent node (inherited)
    2. Update received info based on current node
    3. Process first child, passing needed info
    4. Get back info from child processing (synth.)
    5. Process that info
    6. Process next child, passing needed info
    7. Get back info from child processing
    8. Process that info
    9. Return useful info back to parent node




- Do as many passes as needed

# Factorize common code

VSOP being an expression-based language, nearly everything is an expression, and can be used in any context were an expression is expected.

Try to be **as general as possible** in your code.
*E.g.*, there is likely no need for a separate `compile_cond()` function, just call your `compile_expr()` function. If you need specific checks or operations (checking the type is `bool` or adding a `br` instruction), it should likely go into `compile_if()`. If you want to share condition-checking code between, say if-then-else and while loops, make a `compile_cond()` function, but have it call `compile_expr()` rather than reimplementing the same logic.

**Avoid redundant code!**

# Static vs Dynamic

## Static

- *Static properties* are known at **compile time**
- *Static behaviour* is what your compiler does **when compiling**
- Safer (compile-time error detection)
- More efficient (see later)

## Dynamic

- *Dynamic properties* are only known at **runtime**
- *Dynamic behaviour* is what your compiled program does **when running**
- More expressive

- Before implementing a feature, ask yourself whether it is static, dynamic, or if it needs both static and dynamic support

# Static vs Dynamic Types

```
class Foo { ... }
class Bar extends Foo {
    ...
    public void barOnlyMethod() { ... }
}

class Main {
    public static void main(String[] args) {
        Foo iAmABar = new Bar(); // Static type is Foo
                                 // Dynamic type is Bar
                                 // OK since Foo <: Bar
        iAmABar.barOnlyMethod(); // Always valid, but
                                 // forbidden in VSOP,
    }                            // Java, C++, ...
}                                // but OK in python
```

# Static vs Dynamic values

In general, **values** are only known at run-time, but some values can be determined at compile-time by **constant propagation**, which can lead to **dead-code** elimination.

```
if a == b (* Is the condition true? *)
          (* In general, we don't know until run-time *)
then 42
else 1984;

if true (* Statically known to be true *)
then print("I knew it!")
else print("Oh no! A bug :'("); (* Dead code *)
```

Second case:

- is somewhat rare, but definitely happens (inlining);
- can be optimized away by LLVM anyway (`simplifycfg`).

# KISS: Keep It Simple, Stupid

```
@.str = private unnamed_addr constant [14 x i8] c"Hello, world!\00", align 1

define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 @puts(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture readonly) #1

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false"
"no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

does (roughly) the same as the much simpler

```
@.str = constant [14 x i8] c"Hello, World!\00"

define i32 @main() {
  %1 = getelementptr [14 x i8], [14 x i8]* @.str, i64 0, i64 0
  %2 = call i32 @puts(i8* %1)
  ret i32 0
}

declare i32 @puts(i8*)
```

# Generated Code vs Runtime

- You can generate code inline for most operations, even in a dynamic language. This is efficient but:
  - may lead to code explosion
  - may be hard to do
- You can factorize out common functionalities into a library provided with every VSOP program. This is called the **runtime**
- The runtime can be written in a different, usually lower-level language
- Some languages go as far as using a *virtual machine* to run programs, which are more data than code (e.g. Java, GHC)
- For VSOP, we will provide a runtime for I/O (the `Object` class). Other operations can be generated directly.

# How to check/implement the Object class?

Semantic analysis:

- The fact that `Object` exists and its method prototypes should be known during typechecking.
- Add their *prototypes* manually to your symbol tables, or parse their definitions directly in VSOP (with dummy method bodies).

Code generation:

- Don't generate code for `Object` methods within your compiler.
- Provide the `Object` method definitions separately:
  - by appending their LLVM IR directly to generated IR ;
  - or by linking with an object file with their definitions (*e.g.* generated from C)
- Alternatively, implement the **FFI** extension and write the `Object` class directly in VSOP.

**We will provide C and LLVM IR for the Object class**.

# Accessing the runtime on the platform

Your generated compiler is not run in your `vsopcompiler` folder, so you will not be able to access your `runtime` folder using relative path (`./runtime/`).

**Solution**: Use the makefile to copy the runtime file(s) you need in a known location, eg. `/usr/local/lib/vsop/`. Then your compiler can access it using an absolute path.

# By-value vs By-reference

### By Value

Function arguments are copied before call, modifying an argument inside the callee does not modify corresponding variable at caller site

### By Reference

Function arguments are passed by pointer, modifying an argument inside the callee modifies caller variable

- By-Value is generally safer, modification allowed through explicit pointers
- Like Java, VSOP uses by-value semantics for primitive types (like `int32`), but by-reference semantics for objects.

# How to Compile Method Dispatch?

How would you compile the following?

```
class MyClass {
    j : int32 <- 42;
    someMethod(i : int32) : int32 {
        i + j
    }
}
// ...
    let myObject : MyClass <- new MyClass in
    myObject.someMethod(1942)
```

# Simple static dispatch

Key idea: pass the **object** instance as **first argument** to methods.

```c
// struct to keep fields data
typedef struct { int32_t j; } MyClass;

// Allocation and initialization
MyClass *MyClass_new(void) {
    MyClass *self = malloc(sizeof (MyClass));
    self->j = 42;
    return self;
}

// Methods
int32_t MyClass_someMethod(MyClass *self, int32_t i) {
    return i + self->j;
}

// ...
    MyClass *myObject = MyClass_new();
    MyClass_someMethod(myObject, 1942);
```

# Field inheritance

How to reuse parent field in child method?
How to add new fields to children classes?

```
class Parent {
    i : int32 <- 42;
    j : int32;
}

class Child extends Parent {
    k : int32;
    sum() : int32 { i + j + k }
}
// ...
    let child : Child <- new Child
    in child.sum()
```

# Field inheritance: Parent class

Note that we decouple allocation and initialization.

```
// Parent fields
typedef struct {
  int32_t i;
  int32_t j;
} Parent;

// Initialization of already allocated structure
void Parent_init(Parent *self) {
    self->i = 42;
    self->j = 0;
}

// Allocation and initialization
Parent *Parent_new(void) {
    Parent *self = malloc(sizeof (Parent));
    Parent_init(self); // Initialization
    return self;
}
```

# Field inheritance: Child class

```c
typedef struct {
    int32_t i; // FIRST: fields of parent, IN SAME ORDER
    int32_t j;
    int32_t k; // THEN: additional field(s) of Child
} Child;

void Child_init(Child *self) {
    Parent_init((Parent *) self); // super()
    self->k = 0; }

Child *Child_new(void) {
    Child *self = malloc(sizeof (Child));
    Child_init(self); // Initialization
    return self; }

int32_t Child_sum(Child *self) {
    return self->i + self->j + self->k; }
// ...
    Child *child = Child_new();
    Child_sum(child);
```

# Field inheritance: memory layout

```
%Parent = type { i32, i32 }
%Child  = type { i32, i32, i32 }
```

| Parent | |
|---|---|
| 0 | i |
| 4 | j |

=

| Child | |
|---|---|
| 0 | i |
| 4 | j |
| 8 | k |

# Method inheritance

```
class Parent {
    i : int32 <- 42;
    inParent() : int32 { i }
}

class Child extends Parent {
    inChild() : int32 { 1942 + i }
}

// ...
    let c : Child <- new Child in {
        c.inParent();
        c.inChild()
    }
```

# Method inheritance: static dispatch

Use a `Child` as a `Parent`, using a cast.
Same data layout for common fields of `Child` and `Parent`.

```c
typedef struct { int32_t i; } Parent;
void Parent_init(Parent *self) { self->i = 42; }
Parent *Parent_new(void) { /* ... */ }
int32_t Parent_inParent(Parent *self) { return self->i; }

typedef struct { int32_t i; /* Same as parent */ } Child;
void Child_init(Child *self) { /* ... */ }
Child *Child_new(void) { /* ... */ }
int32_t Child_inChild(Child *self) { return 1942 + self->i; }

// ...
    Child *c = Child_new();
    // c.inParent()
    Parent_inParent((Parent *) c); // Cast needed (and OK) here
    // c.inChild()
    Child_inChild(c);
```

# Static dispatch is not enough

```
class Person {
    i : int32;
    name() : string { "Someone" }
}

class John extends Person {
    name() : string { "John" }
}

class MyClass {
    someMethod(p : Person): unit {
        print(p.name()); // What to call here?
        // Person_name(), John_name(), Mary_name(), ...?
        ()
    }
}
```

# Method overridding: function pointers as fields

```
typedef struct Person {
    int32_t i;
    char *(*_name)(struct Person *);
} Person;
char *Person_name(Person *self) { return "Someone"; }
void Person_init(Person *self) {
    self->i = 0;
    self->_name = &Person_name;
}

typedef struct John {
    int32_t i; // Same as Person
    char *(*_name)(struct John *); // John instead of Person, is it safe?
} John;
char *John_name(John *self) { return "John"; }
void John_init(John *self) {
    Person_init((Person *) self);
    self->_name = &John_name; // Overrides name()
}
// ...
    p->_name(p) // p.name()
```

# Method overridding: function pointers as fields

Using fields for methods works, but is very **wasteful**.

All objects (*i.e.* instances) of the same class share the same set of methods, but every single one of them carries one pointer per method. **One pointer per method per object!**

Idea: share the method pointers between objects of the same class. We need to associate each object with a table of method pointers.

Either use **fat pointers** (*i.e.* a pointer to the object + a pointer to the table of method pointers), or keep a pointer to the table in the object itself (the **vtable** pointer).

# Method Dispatch: Types

```c
// Type for object instances
typedef struct {
    // Virtual function table (see below)
    struct MyClassVTable *vtable;
    // Fields (each object instance needs its own)
    int32_t j;
} MyClass;

// Type for the methods (type of virtual function table)
struct MyClassVTable { // Types for methods of MyClass
    void (*someMethod)(MyClass *, bool);
    int32_t (*someOtherMethod)(MyClass *, int32_t);
};
```

# Method Dispatch: Constructor and Methods

```c
void MyClass_init(MyClass *self) {
    // Initialize fields, including virtual function table
    self->vtable = &MyClass_vtable;
    self->j = 42; // j : int32 <- 42
}

// someMethod(b : bool) : unit { ... }
void MyClass_someMethod(MyClass *self, bool b) { /* ... */ }

// someOtherMethod(i : int32) : int32 { i + j }
int32_t MyClass_someOtherMethod(MyClass *self, int32_t i) {
    return i + self->j;
}

// Actual function table object (only one needed)
struct MyClassVTable MyClass_vtable = {
    .someMethod = &MyClass_someMethod,
    .someOtherMethod = &MyClass_someOtherMethod
};
```

# Method Dispatch: Calling a Method

```
// let myObject : MyClass <- new MyClass in
MyClass *myObject = MyClass_new();
// myObject.someOtherMethod(42)
myObject->vtable->someOtherMethod(myObject, 42);
```

Note the cost of vtable-based dispatch: each method call now requires **two memory accesses**:

- one to get the vtable pointer;
- one to get the method pointer.

One can optimize out virtual calls when the actual type of the object is known (*e.g.* just after `new`).
In loops, one can also cache the method pointer.

# Method Dispatch: Adding Inheritance to the Mix

```
class Parent { (* ... *) }
class Child extends Parent { (* ... *) }

class X {
    someMethod(Parent p) : unit {
        // Static type of p is Parent, but dynamic type?
        p.overridenMethod()
    }
}
```

```
void X_someMethod(X *self, Parent *p) {
    /* Will find the right method, whether p is a Parent,
     * a Child or any other descendant of Parent */
    p->vtable->overriddenMethod(p);
}
```

# Method Dispatch: Adding Inheritance to the Mix (Cnt'd)

```
class Parent {
    inheritedField : int32 <- 42;
    inheritedMethod() : unit { (* ... *) }
    overriddenMethod() : unit { (* ... *) }
}

class Child extends Parent {
    newField : bool <- true;
    newMethod() : unit { (* ... *) }
    overriddenMethod() : unit { (* ... *) }
}
```

# Method Dispatch: Adding Inheritance to the Mix (Cnt'd)

```c
// A Child must be able to masquerade as a Parent
typedef struct {
    struct ChildVTable *vtable; // Virtual function table first
    int32_t inheritedField; // Parent fields, in the same order as parent!
    bool newField; // And, finally, Child's new fields
} Child;

// A ChildVTable must be able to masquerade as a ParentVTable
struct ChildVTable {
    // First, parent methods in the same order
    void (*inheritedMethod)(Child *), // Why not Parent * here?
    void (*overriddenMethod)(Child *),
    // Then child's new methods
    void (*newMethod)(Child *)
}

// Child VTable can mix inherited, overridden and new methods
struct ChildVTable Child_vtable {
    // Necessary (but legit) cast for inherited method
    .inheritedMethod = (void (*)(Child *)) Parent_inheritedMethod,
    .overriddenMethod = Child_overriddenMethod,
    .newMethod = Child_newMethod
}
```

# Method Dispatch: Adding Inheritance to the Mix (Cnt'd)

Chain constructors properly!

```
void Child_init(Child *self) {
    Parent_init((Parent *) self); // Parent initializers
    self->vtable = &Child_vtable; // Override vtable
    self->newField = true;
}

Child *Child_new() {
    Child *self = malloc(sizeof (Child));
    Child_init(self);
    return self;
}
```

# In summary

| Parent | |
|---|---|
| 0 | vtable |
| 8 | inheritedField |

| Child | |
|---|---|
| 0 | vtable |
| 8 | inheritedField |
| 12 | newField |

| ParentVTable | |
|---|---|
| 0 | inheritedMethod() |
| 8 | overriddenMethod() |

| ChildVTable | |
|---|---|
| 0 | inheritedMethod() |
| 8 | overriddenMethod() |
| 16 | newMethod() |

# Beware

What if my class has a field named `vtable`?

What if my class has a method called `init`?

What if I have both:

- a class named `Base` with a method `jump_ship()`;
- a class named `Base_jump` with a method `ship()`?

# Beware of Symbol Conflicts

What if my class has a field named `vtable`?

What if my class has a method called `init`?

What if I have both:

- a class named `Base` with a method `jump_ship()`;
- a class named `Base_jump` with a method `ship()`?

⇒ Symbol conflicts!

Use **name mangling** to avoid all possible conflicts in generated symbol names. For example:

- `_vtable` field (fields cannot start with `_`).
- `ClassName__methodName` (methods cannot begin with `_`).
- `ClassName___init` and `ClassName___vtable` instance (`new` is a keyword).

# Outline

# Questions and (possibly) answers

?