Compile-Time Function Evaluation (CTFE)

Josef Podaný, 2.3.2023

Talk Structure

1. Introduction to CTFE, motivation
2. Case Study
    1. C++ *constexpr* & Clang: walking AST or interpreting byte-code
    2. Zig *comptime*: interpreting untyped ZIR
3. Conclusion
    1. Benefits & Drawbacks
    2. Remarks

Introduction

What Is CTFE?

Introduction

- Extension of optimizations called **constant propagation/folding**
- Present in C++, Rust, Zig...

"In computing, compile-time function execution is the ability of a compiler, that would normally compile a function to machine code and execute it at runtime, to execute the function at compile time." [1]

Why Is CTFE Useful?

Introduction

- Precomputing values ⇒ **writing fast**(er) **code**
- Static checks & assertions ⇒ **writing robust code**

Case Study: C++ & Clang

C++ *constexpr* Specifier

Case Study: C++ & Clang

- The "precise" definition of *constexpr* in C++ is cumbersome, informally...
  - *constexpr* applies to variables and functions,
  - and can be computable at compile-time.
- Still evolving through newer standards
- If the compiler meets *constexpr* when parsing...
  - it must be able to evaluate the expression immediately,
  - and whenever undefined behavior is met, the compiler MUST fail
- Related only to the front end of the compiler

Quick Quiz

Case Study: C++ & Clang

Will this compile?

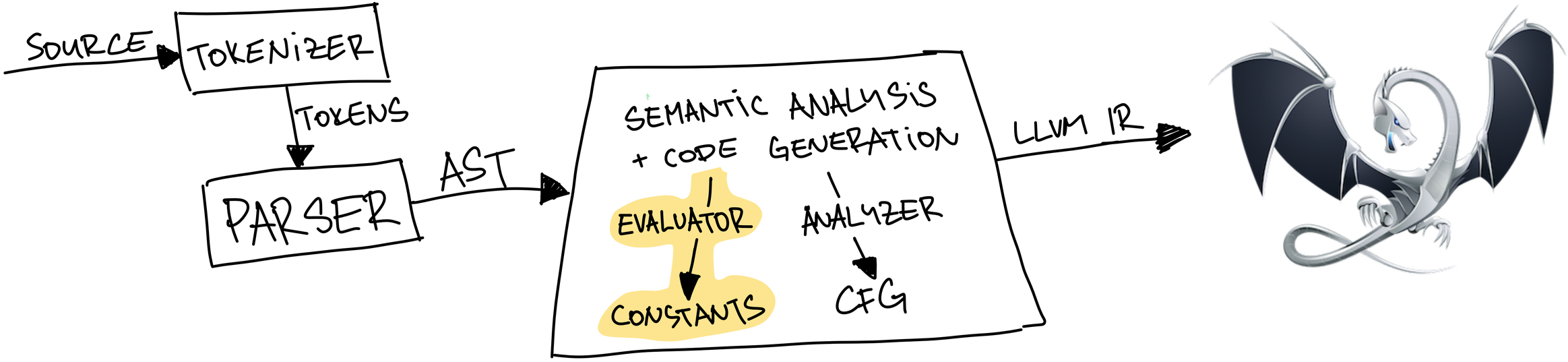
```
constexpr int foo(int i );

void bar() {
    constexpr auto X = foo(3);
}

constexpr int foo(int i ) {
     return 2 * i;
}
```

High-Level Clang Architecture

Case Study: C++ & Clang



Abstract Syntax Tree Evaluation

Case Study: C++ & Clang

- Idea: Recursively walk & evaluate the AST if constexpr node is met
- Main entry function *Expr::Evaluate* implemented in *AST/ExprConstant.cpp*

```
static bool Evaluate(APValue &Result, EvalInfo &Info, const Expr *E) {
    QualType T = E->getType();
    if (E->isGLValue() || T->isFunctionType()) {
        LValue LV;
        if (!EvaluateLValue(E, LV, Info))
            return false;
        LV.moveInto(Result);
    } else if ( ... ){
        // Same pattern goes on for other types of expressions
    }

    return true;
}
```

Constant Interpreter

Case Study: C++ & Clang

- Idea:
  1. Translate *constexpr* AST subtree into byte-code
  2. Interpret the byte-code
- Advantageous when evaluating loops and same function multiple times
- Can be enabled with flag *-fexperimental-new-constant-interpreter*

Constant Interpreter Byte-code

Case Study: C++ & Clang

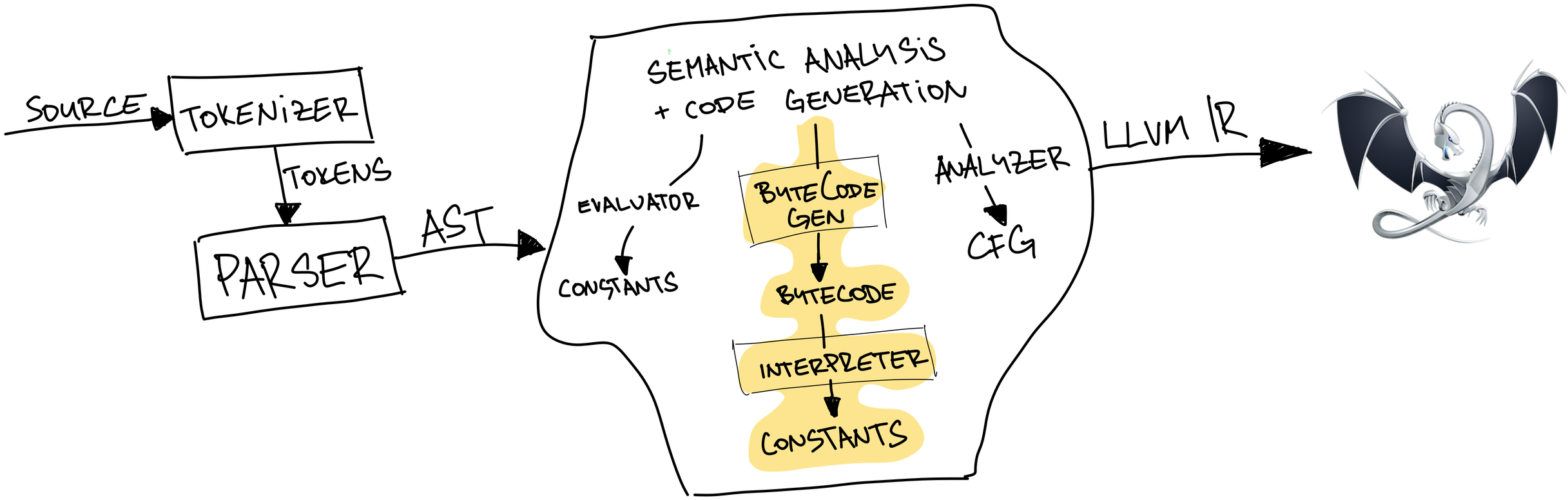
```
acc = acc + ptr[i]
```

```
GetLocalSint32 a
GetParamPtr ptr
GetLocalSint32 i
AddOffsetSint32
LoadSint32
AddSint32
SetLocalSint32 a
```

High-Level Clang Architecture II

Case Study: C++ & Clang

- Including the **constant interpreter** into the picture:



Constant Interpreter Cont.

Case Study: C++ & Clang

- On given cases, performance is 10x better than when walking the AST
- Constant interpreter covers ~70% of C++11 features
- Development was halted until November 2022

Interesting Implications of *constexpr*

Case Study: C++ & Clang

- Detection of memory leaks in transient allocation, accessing memory outside of constexpr via pointers
  - Must solve some kind of memory sanitization, pointer arithmetic

Case Study: Zig

Quick Introduction To Zig

Case Study: Zig

*"Zig is a general-purpose programming language and toolchain for maintaining **robust**, **optimal** and **reusable** software."*

- Manual memory management
- No hidden allocations
- CTFE more flexible than in C++

What Can Zig Do

Case Study: Zig

```zig
// Compares two strings ignoring case (ascii strings only).
// Specialzied version where `uppr` is comptime known and *uppercase*.
fn insensitive_eql(comptime uppr: []const u8, str: []const u8) bool {
    comptime {
        var i = 0;
        while (i < uppr.len) : (i += 1) {
            if (uppr[i] >= 'a' and uppr[i] <= 'z') {
                @compileError("`uppr` must be all uppercase");
            }
        }
    }

    var i = 0;
    while (i < uppr.len) : (i += 1) {
        const val = if (str[i] >= 'a' and str[i] <= 'z')
            str[i] - 32
        else
            str[i];
        if (val != uppr[i]) return false;
    }

    return true;
}

pub fn main() void {
    const x = insensitive_eql("Hello", "hElLo");
}
```

```zig
const std = @import("std");

pub fn main() !void {
    const comptime_res = comptime add(32, 10);
    const runtime_res = add(32, 10);

    std.log.info("comptime result: {}", .{comptime_res});
    std.log.info("runtime result: {}", .{runtime_res});
}

fn add(fst: u8, snd: u8) u8 {
    return fst + snd;
}
```

CTFE in Zig

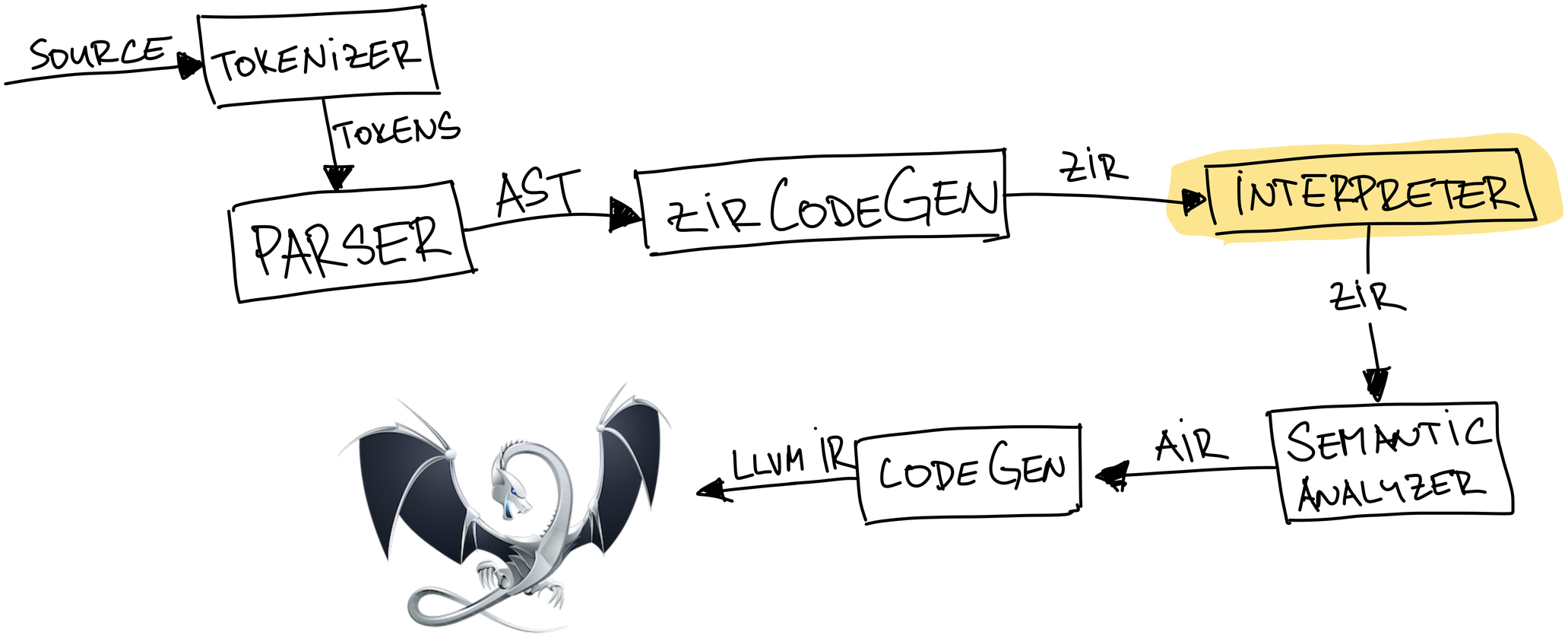
Case Study: Zig

- Implemented the constant interpreter from the get-go
- Generics implemented via *comptime* type variables (monomorphization)

```
fn List(comptime T: type) type {
    return struct {
        items: []T,
        len: usize,
    };
}
```

Zig Compiler Architecture

Case Study: Zig



ZIR: Zig Internal Representation

Case Study: Zig

```
const t = bool;
const result: t = 42;
```

```
%0 = extended(struct_decl(parent, Auto, {
    [16] t line(0) hash(243086356f9a6b0669ba4a7bb4a990e4): %1 = block_inline({
        %2 = break_inline(%1, @Ref.bool_type)
    }) node_offset:1:1
    [24] result line(1) hash(8c4cc7d2e9f1310630b3af7c4e9162bd): %3 = block_inline({
        %4 = decl_val("t") token_offset:2:10
        %5 = as_node(@Ref.type_type, %4) node_offset:2:10
        %6 = int(42)
        %7 = as_node(%5, %6) node_offset:2:14
        %8 = break_inline(%3, %7)
    }) node_offset:2:1
}, {}, {})
```

Conclusion

We've Showed...

Conclusion

- Two "simple" approaches: AST walking and interpretation
- **Abstract Syntax Tree Evaluation**
  - Define "*walking functions*" for each node type in the AST
  - Recursively traverse the AST
- **Virtual Machines & Interpreters**
  - Compile AST to some IR
  - Optionally, optimize the IR
  - Interpret the IR & add the result to the corresponding AST node

Final Remarks

Conclusion

- Although the idea behind constant interpreter is pretty simple, *correct* implementation is hard
- Documentation is almost non-existent!
  - Found 1 talk from LLVM, a few Clang docs, and no GCC docs
  - C++ design docs do not hint at how to implement features
- Clang constant interpreter is still experimental
- Getting insight from the source code of compilers is ... really hard

Thank You! Questions?

Links

- https://en.wikipedia.org/wiki/Compile-time_function_execution
- https://clang.llvm.org/docs/ConstantInterpreter.html
- https://www.youtube.com/watch?v=LgrgYD4aibg
- https://www.redhat.com/en/blog/new-constant-expression-interpreter-clang
- https://clang.llvm.org/docs/ConstantInterpreter.html
- https://pvs-studio.com/en/blog/posts/cpp/0909/
- https://clang.llvm.org/docs/InternalsManual.html#constant-folding-in-the-clang-ast
- https://clang.llvm.org/doxygen/ExprConstant_8cpp.html#ae7dcbcdd6d581c905cc4eeb5fa79c86b
- https://en.cppreference.com/w/cpp/language/constant_expression
- https://mitchellh.com/zig/astgen
- https://www.youtube.com/watch?v=8MbREuiLQrM
- https://andrewkelley.me/post/zig-programming-language-blurs-line-compile-time-run-time.html