Higgs, an Experimental JIT Compiler written in D

DConf 2013

Maxime Chevalier-Boisvert

Université de Montréal

Introduction

- PhD research: compilers, optimizing dynamic languages, type analysis, JIT compilation
- Higgs: experimental optimizing JIT for JS
- The core of Higgs is written in D
- This talk will be about
  - Dynamic language optimization
  - Higgs, JIT compilation, my research
  - Experience implementing a JIT in D
  - A JIT for D's CTFE

Dynamic Languages

- Dynamic typing
  - Types associated with values
  - Variables can change type over time
  - No type annotations
- Late binding
  - Symbols resolved dynamically (e.g.: globals)
- Dynamic loading of code (eval, load)
- Dynamic growth of objects
  - Objects as dictionaries

Why so Slow?

- Reputation for being slow
  - Easiest to implement in an interpreter
  - Naive implementations have big overhead
- Values are usually “boxed”
  - Values as pairs: datum + type tag
  - Values as objects: CPython's numbers
- Basic operators (+, -, *, ...) have dynamic dispatch
- Global and field accesses as hash table lookups

Making it Fast

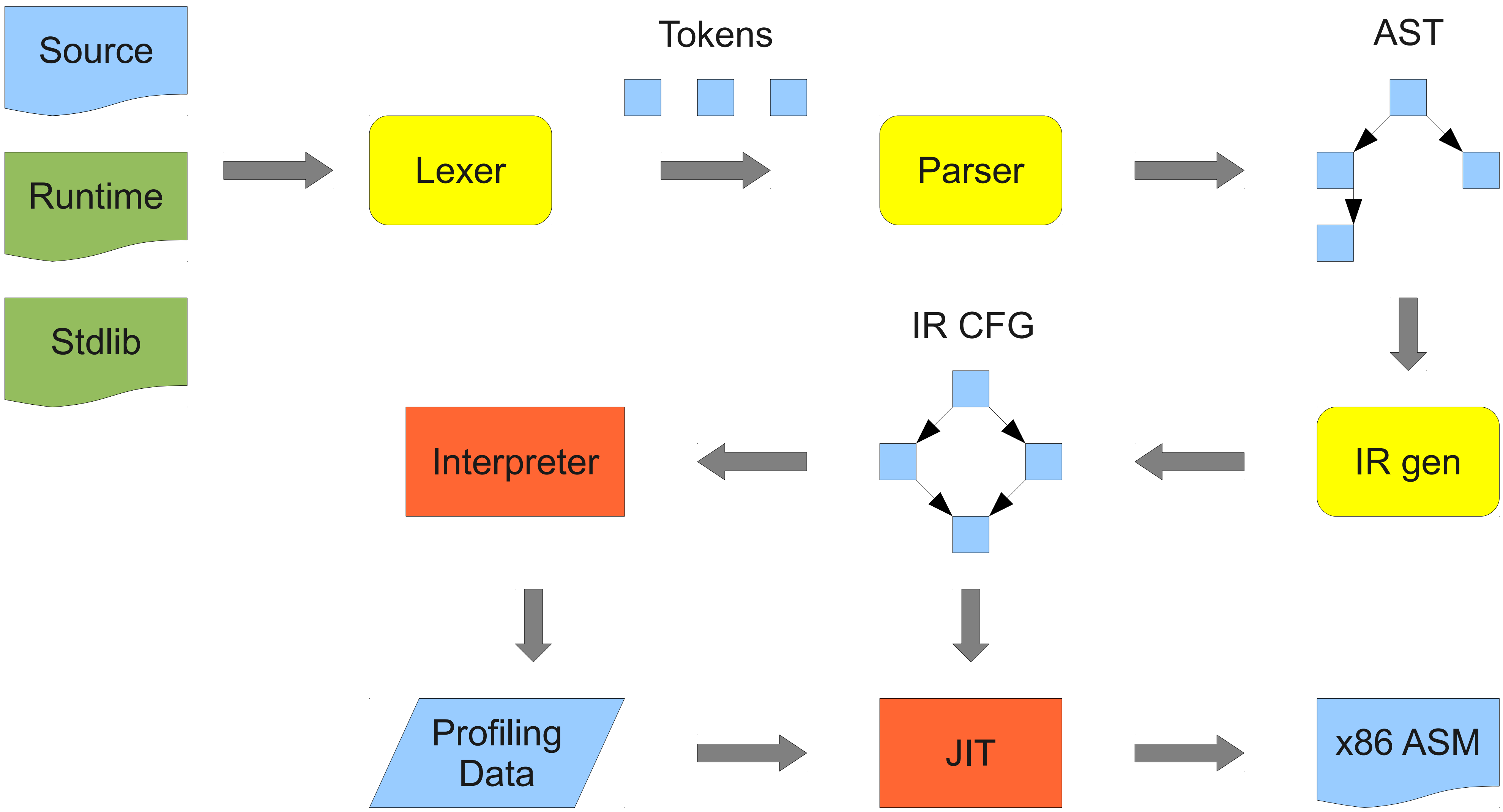
- Make the code more static
  - Remove dynamic behavior where possible
- Requires type information
  - Profiling
  - Type analysis
- Prove that specific variables have a given type
  - e.g.: x is always an integer
  - e.g.: the function foo will never be redefined

Harder than it seems

- JS, Python, Ruby not designed with performance in mind
  - Python: (re)write critical parts in C
- Dynamic code loading, `eval`
  - Can break your assumptions
- Numerical towers, overflow checks
  - Hard to prove overflows won't happen

Higgs

- Two main components:
  - Interpreter
  - JIT compiler
- Moderate complexity:
  - D: ~23 KLOC
  - JS: ~11 KLOC
  - Python: ~2 KLOC
- JS support:
  - ~ES5, no property attributes, no `with`

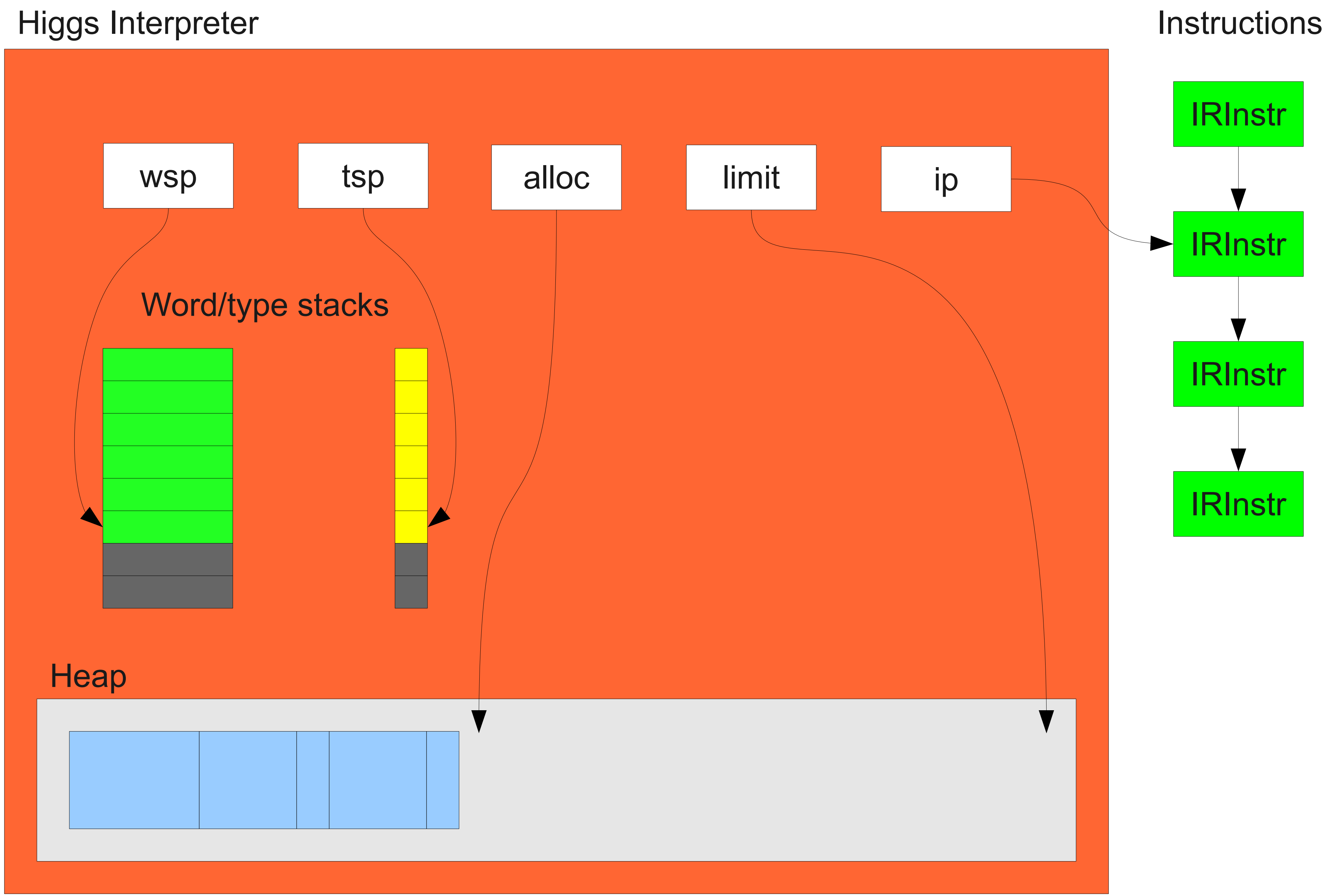

Source
Runtime
Stdlib
Lexer
Tokens
Parser
AST
IR gen
IR CFG
Interpreter
Profiling Data
JIT
x86 ASM

Building Higgs

- Lexer and parser written from scratch, in D
- Designed IR, began implementing AST->IR
- Began implementing basic interpreter
- Grew interpreter, runtime to cover more JS
- Built an x86 assembler, in D
- Implemented basic JIT compiler
- Currently:
  - Implementing research ideas into JIT
  - Icing on the cake: FFI, library support
- Added new unit tests at every step

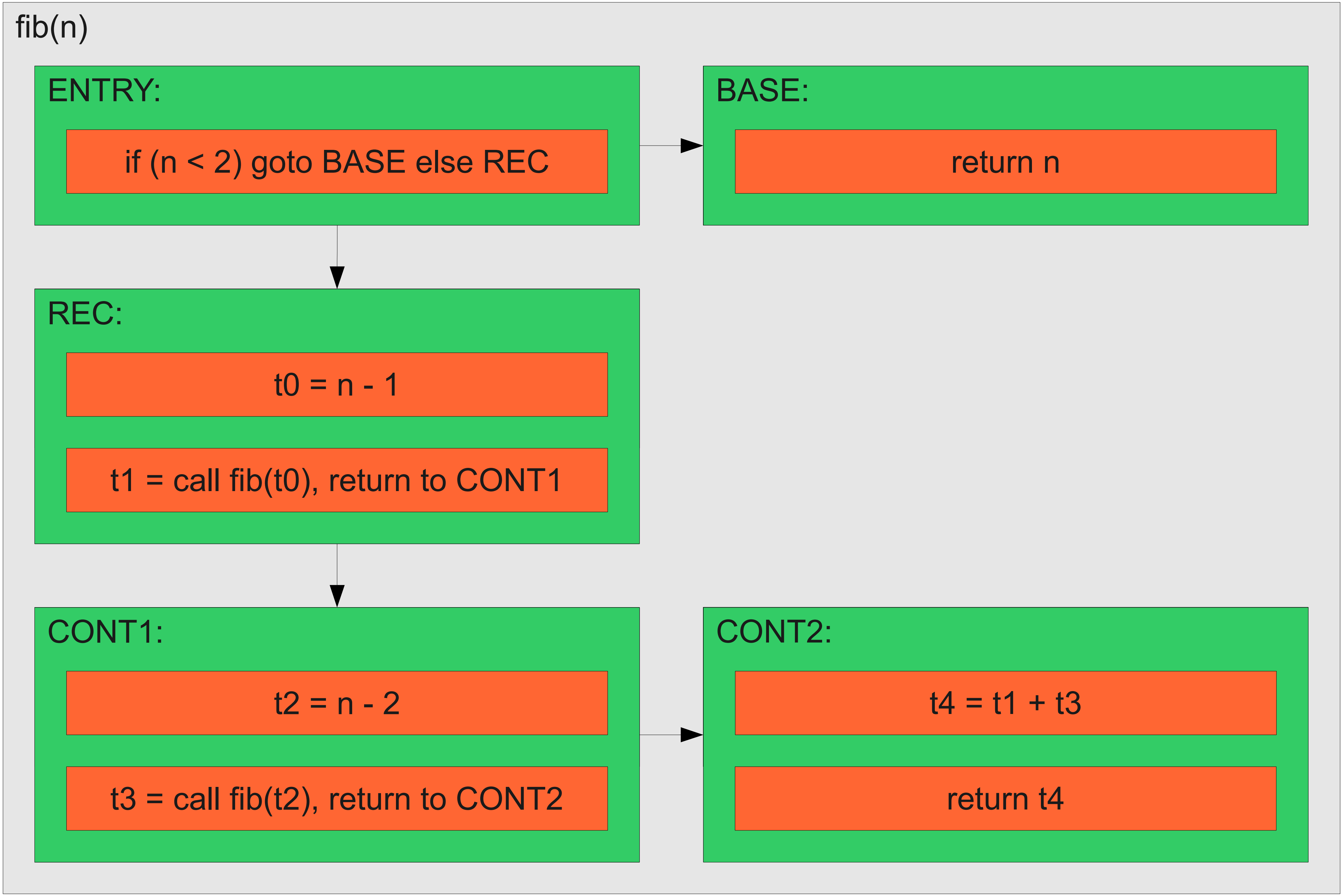
The Interpreter

- Interpreter is used:
  - For profiling
  - Fallback for unimplemented JIT features
  - To start executing code faster
- Designed to be:
  - Simple, easy to maintain
  - Quick to extend and experiment with
  - "JIT-friendly"
- Interpreter is quite slow, 1000 cycles/instr


Higgs Interpreter
Instructions
IRInstr
IRInstr
IRInstr
IRInstr
wsp
tsp
alloc
limit
ip
Word/type stacks
Heap

JIT-Friendly

- Register based VM, not stack-based
  - Easier to analyze/optimize
- IR based on a control-flow graph, not AST
  - Closer to machine code
  - Easier to reason about
- Interpreter stack is an array of values/words
  - Directly reused by the JIT
- Not recursive


fib(n)
ENTRY:
if (n < 2) goto BASE else REC
BASE:
return n
REC:
t0 = n - 1
t1 = call fib(t0), return to CONT1
CONT1:
t2 = n - 2
t3 = call fib(t2), return to CONT2
CONT2:
t4 = t1 + t3
return t4

Low-level Instructions

- Higgs interprets a low-level IR
- Simplifies the interpreter
  - Deals with simple, low-level ops
    - e.g.: imul, fmul, load, store, call, ret
  - Knows little about JS semantics
- Simplifies the JIT
  - Less duplicated functionality in interpreter and JIT
  - Avoids implicit dynamic dispatch in IR ops
    - e.g.: the + operator in JS has lots of implicit branches!

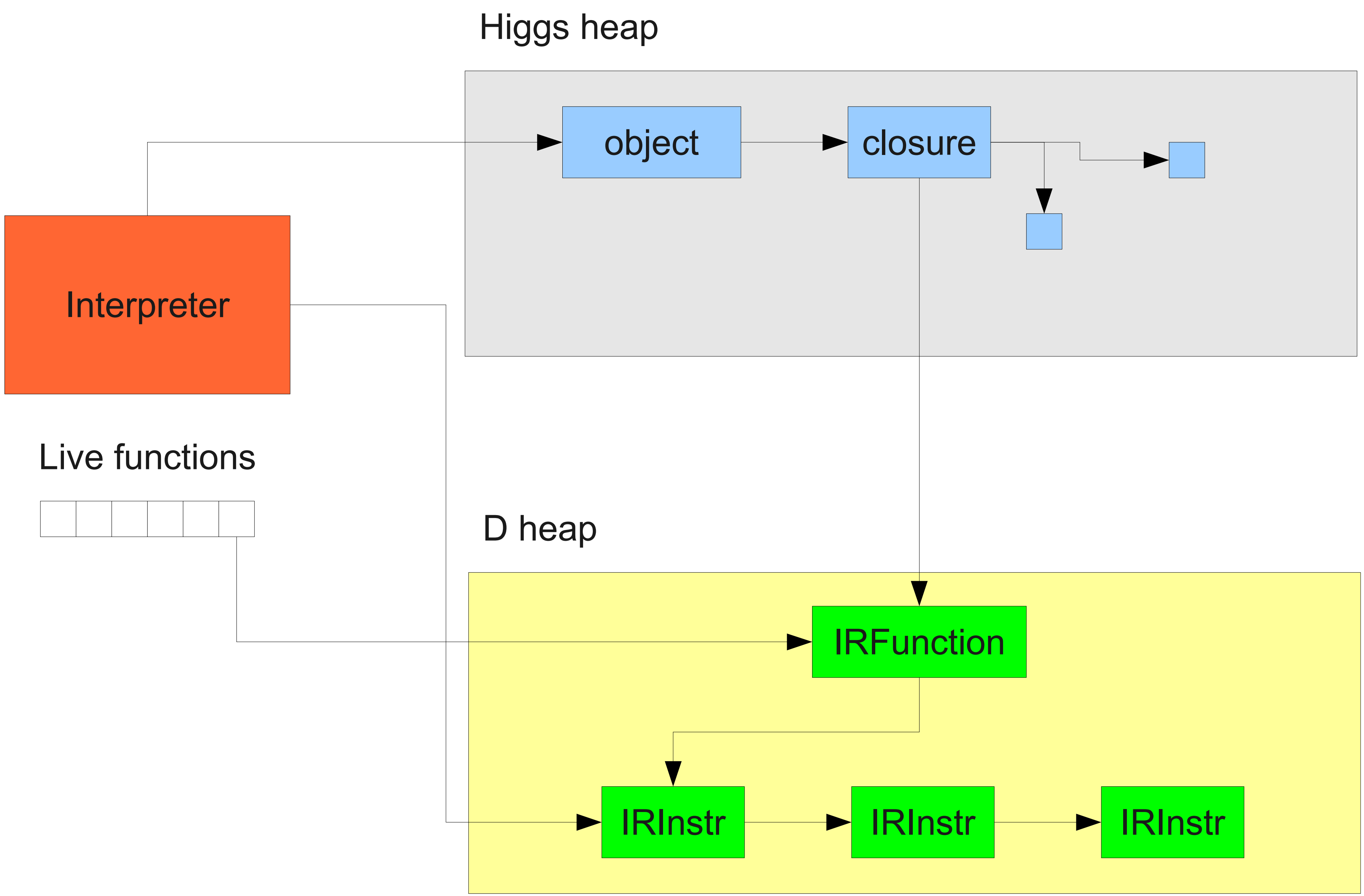
Self-hosting

- Runtime and standard library are self-hosted
- JS primitives (e.g.: JS add operator) are implemented in an extended dialect of JS
  - Exposes low-level operations
- Primitives are compiled/inlined/optimized like any other JS code
  - Avoids opaque calls into C or D code
- Easy to extend/change runtime
- Higher compilation times
- Inlining is critical

```
// JS less-than operator (x < y)
function $rt_lt(x, y)
{
    // If x is integer
    if ($ir_is_int32(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_i32(x, y);

        if ($ir_is_float(y))
            return $ir_lt_f64($ir_i32_to_f64(x), y);
    }

    // If x is float
    if ($ir_is_float(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_f64(x, $ir_i32_to_f64(y));

        if ($ir_is_float(y))
            return $ir_lt_f64(x, y);
    }

    …
}
```

The Higgs Heap

- Higgs manages its own heap for JS objects
- GC is copying, semi-space, stop-the-world
  - Extremely simple
  - Allocation by incrementing a pointer
- References to D objects must be maintained
  - i.e.: Function IR/AST
- Interpreter manipulates references to JS heap
  - Higgs GC might invalidate these


Higgs heap
object
closure
Interpreter
Live functions
D heap
IRFunction
IRInstr
IRInstr
IRInstr

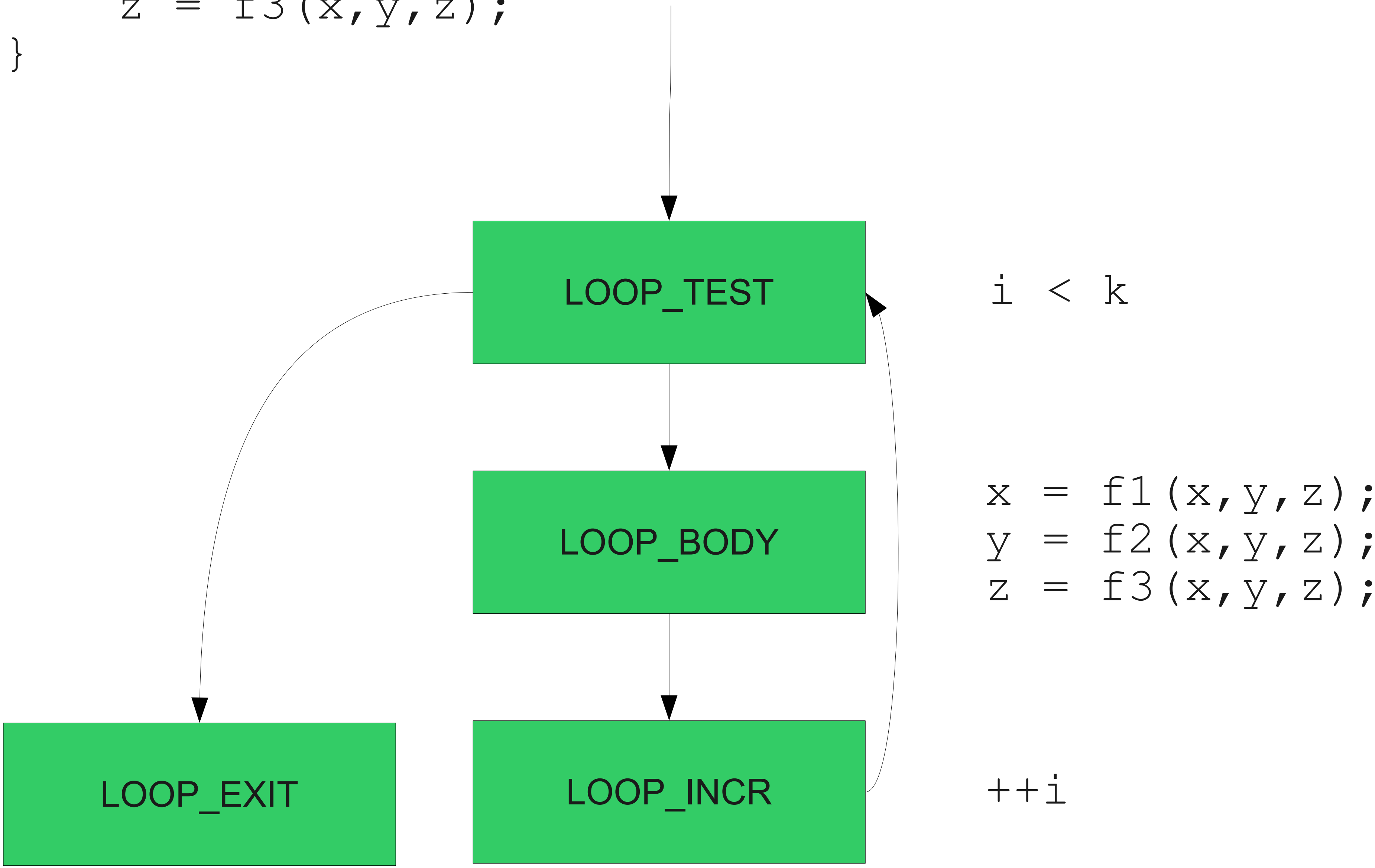
The JIT Compiler

- Targets x86-64 only, for simplicity
- Kicks in once functions have been found hot enough (worth compiling)
  - Execution counters on basic blocks
- Currently fairly basic
  - No inlining, bulk of code is function calls
- Speedups of 5 to 20x
  - Expected to soon reach 100x+ speedups

Current Research

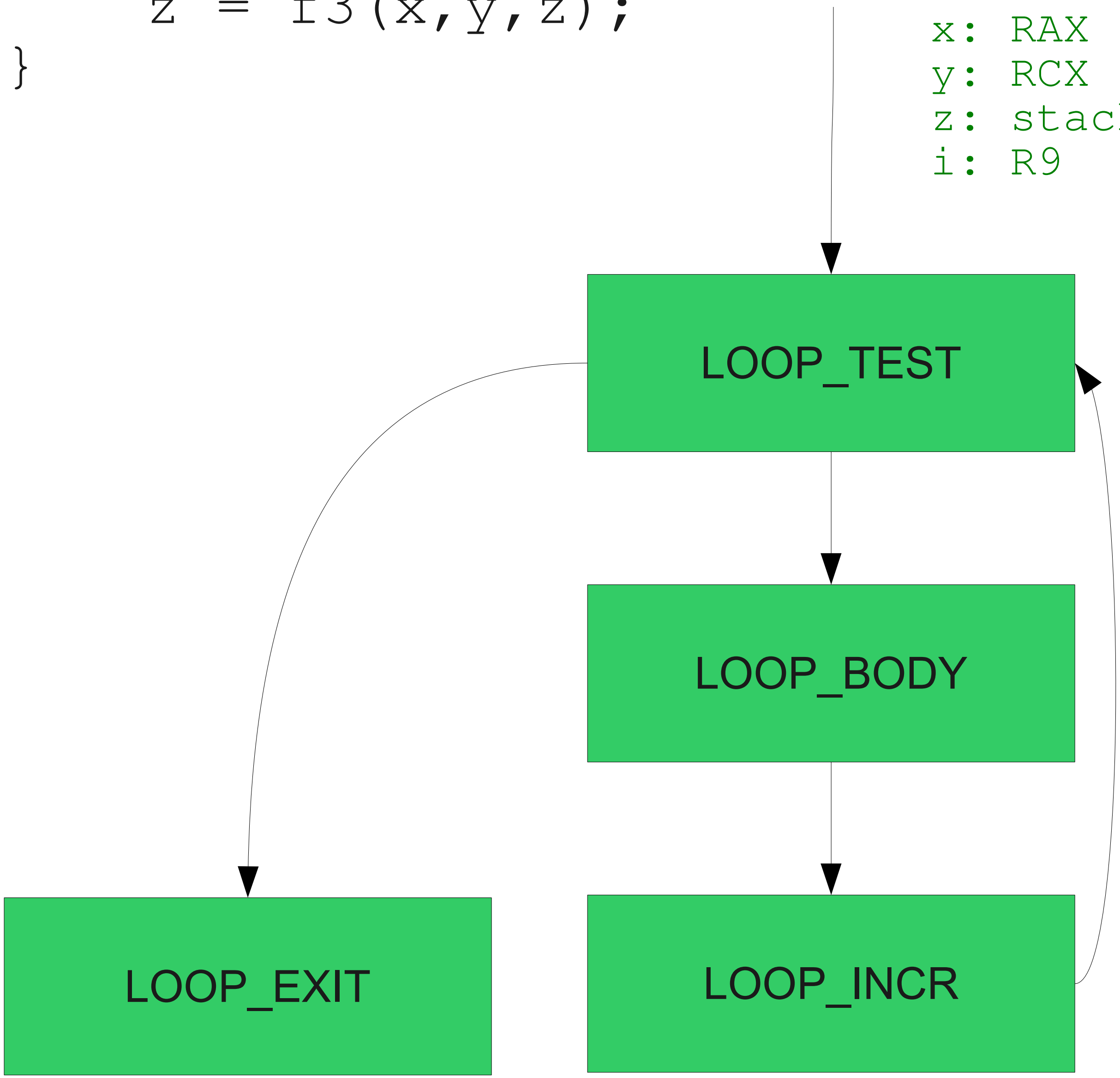
- Context-driven basic block versioning
  - Similar idea to procedure cloning
- Specializing based on:
  - Low-level type information
  - Register allocation state
  - Accumulated facts
- Integrating this in the JIT
- Similarities with trace compilation

```
for (i = 0; i < k; ++i) {
    x = f1(x,y,z);
    y = f2(x,y,z);
    z = f3(x,y,z);
}
```

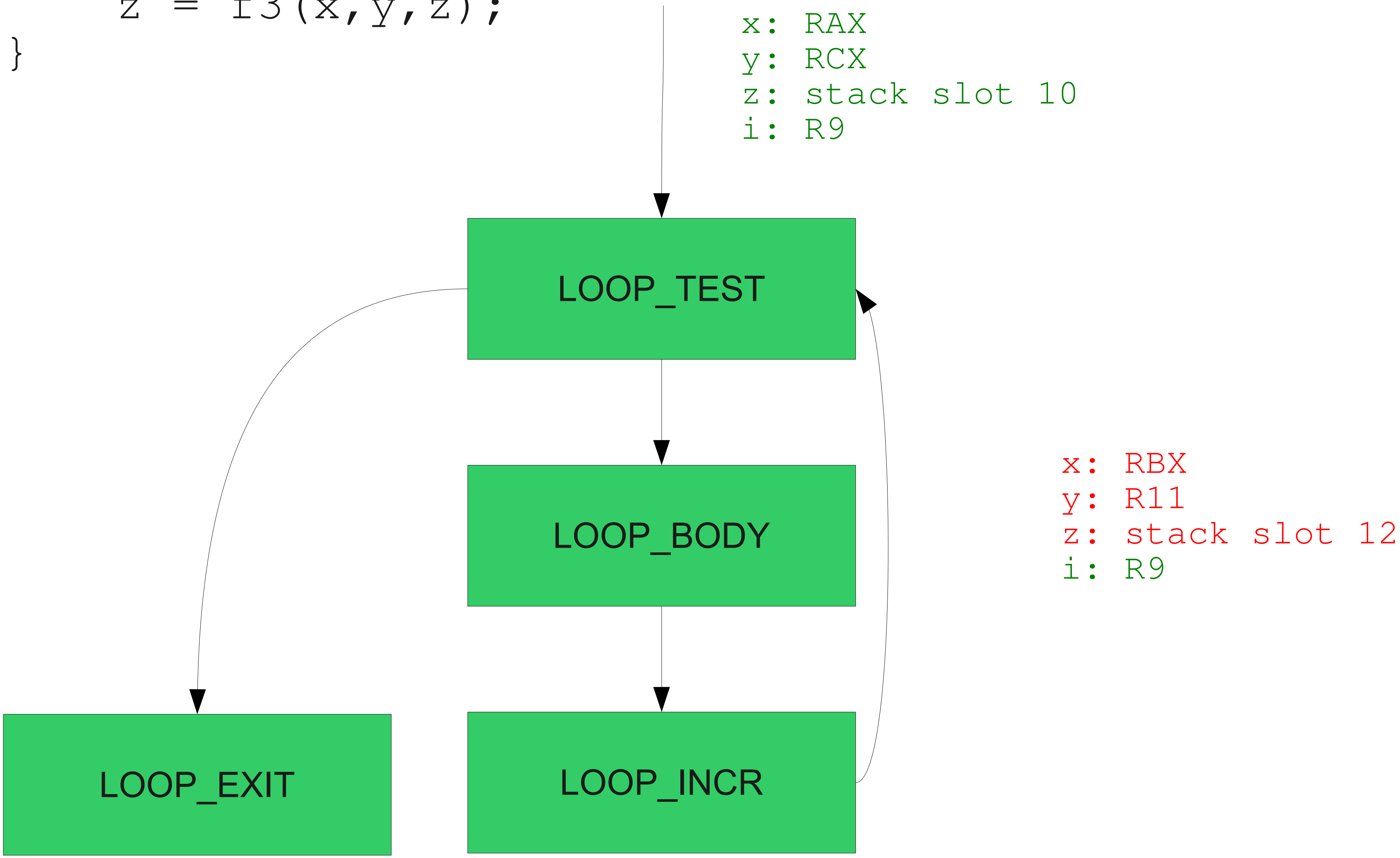


```
for (i = 0; i < k; ++i) {
    x = f1(x,y,z);
    y = f2(x,y,z);
    z = f3(x,y,z);
}
```

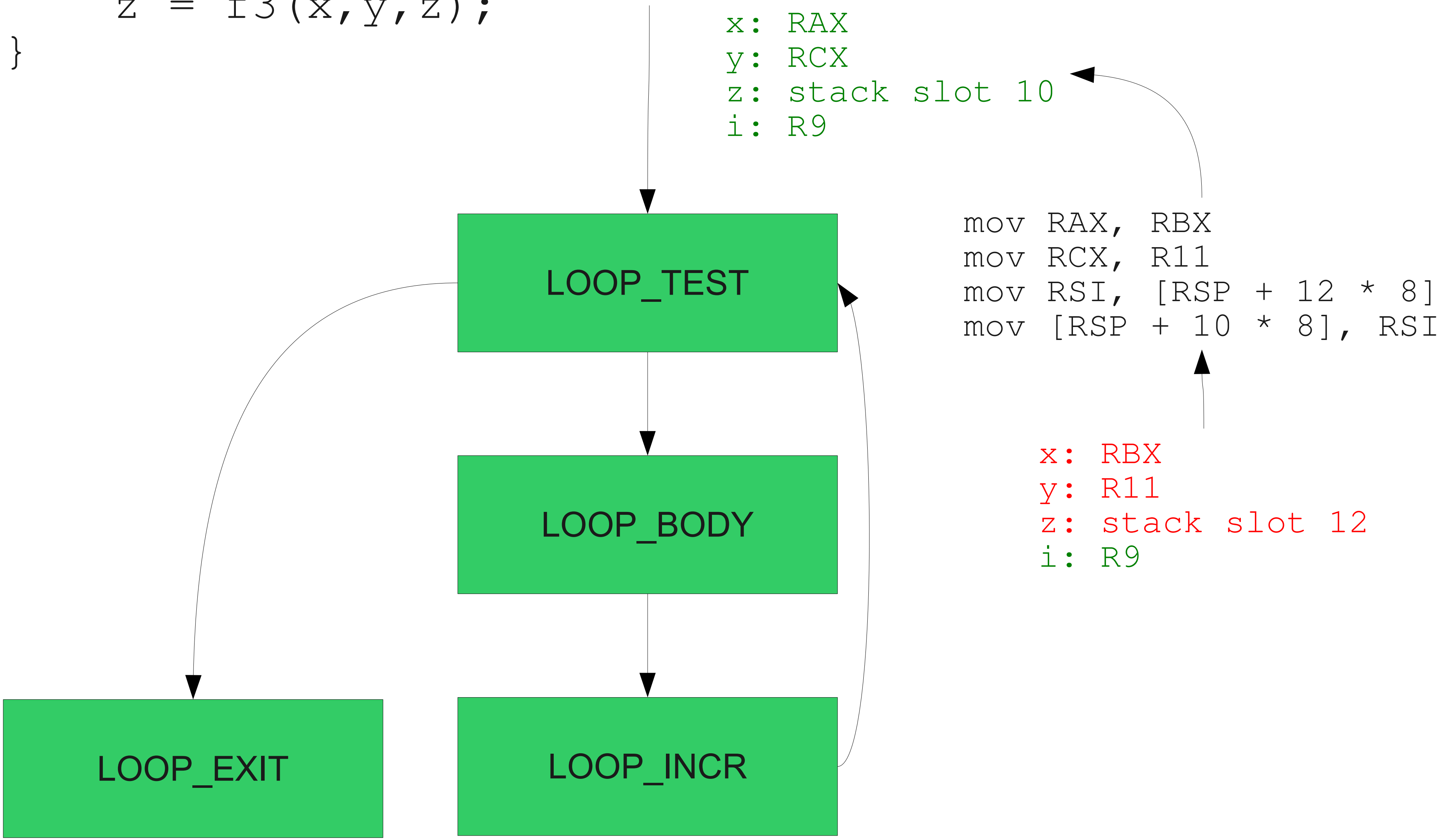
x: RAX
y: RCX
z: stack slot 10
i: R9

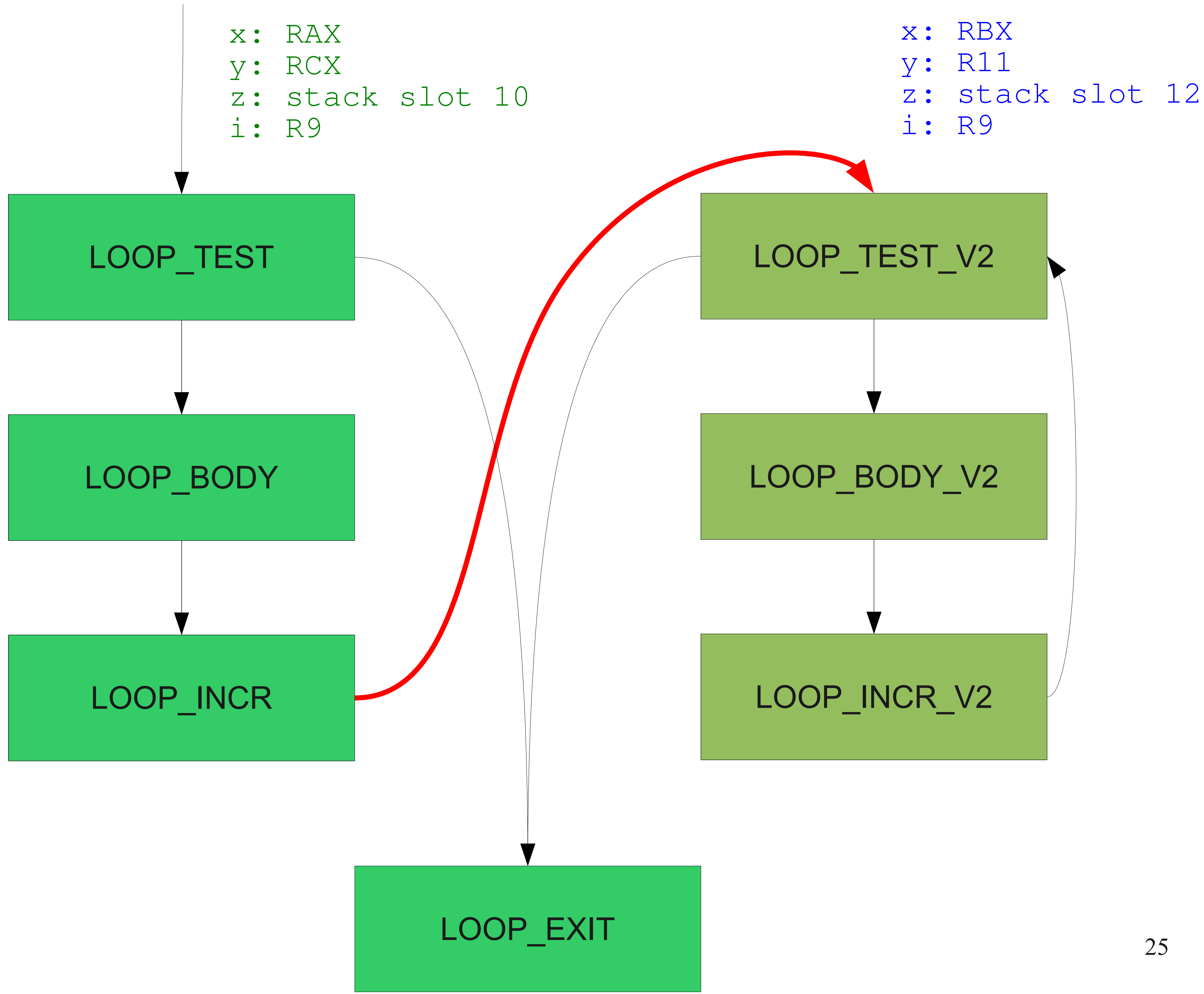


```
for (i = 0; i < k; ++i) {
    x = f1(x,y,z);
    y = f2(x,y,z);
    z = f3(x,y,z);
}
```



```
for (i = 0; i < k; ++i) {
    x = f1(x,y,z);
    y = f2(x,y,z);
    z = f3(x,y,z);
}
```




x: RAX
y: RCX
z: stack slot 10
i: R9
x: RBX
y: R11
z: stack slot 12
i: R9
LOOP_TEST
LOOP_BODY
LOOP_INCR
LOOP_TEST_V2
LOOP_BODY_V2
LOOP_INCR_V2
LOOP_EXIT

Advantages

- Automatically do loop peeling (when useful)
- Automatically do tail duplication
- Register allocation
  - Fewer move operations
  - Make simpler allocators more efficient
- Similar to trace compilation
  - Accumulate knowledge
  - Specialize based on types, constants

A “Multi-world” View

- Traditional control-flow analysis
  - Compute a fixed-point (LFP or GFP)
  - At each basic block, solution must agree
  - Pessimistic answer agrees with all inputs
- Block versioning
  - Multiple solutions possible for a block
  - Don't necessarily have to sacrifice
  - Shifting fixed point to versioning of blocks

Research Questions

- How much code blowup can we expect?
  - Will we have to limit block versioning?
  - What can we do to reduce code blowup?
- What performance gains can we expect?
- What kind of info should we version with?
  - Constant propagation
  - Granularity of type info used
  - How much is too much?
- What is the effect on compilation time?

Why did you choose D?

JIT Compilers

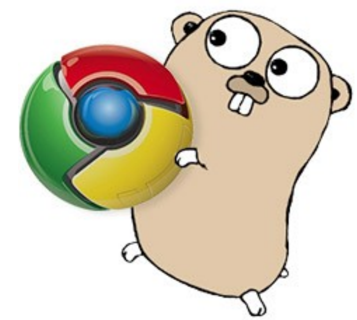
- Need access to low-level operations
  - Manual memory management
  - Raw memory access
  - System libraries
- Are very complex pieces of software
  - Pipeline of code transformations
  - Several interacting components
- Want to mitigate complexity
  - Expressive language
  - Garbage collection

I like C++, but...

- C++ is very verbose
- Header files are frustrating
  - Redundant declarations
  - Poor organization of code
  - Annoying constraints
- C macros are messy and weak
- C++ templates still feel limited
- No standard GC implementation

Other Options

- Google's Go
  - No templates/generics
  - No pointer arithmetic (without casting)
  - Very minimalist and very opinionated



- Mozilla's Rust
  - Very young, still in flux
  - Not an option when I started

D to the rescue!

- Garbage collection by default
  - But manual memory management is still possible
- Has been around for over a decade
  - More mature than newer systems languages
- Attractive collection of features
  - mixins, CTFE, templates, closures
  - Freedom to choose
- Community is active, responsive

Learning D

- If you know C++, you can write D code
  - Similar enough, easy adaptation
  - Slightly less verbose
  - It's actually easier
- Most of the adaptation is learning new idioms
  - Better/simpler ways of doing certain things
- Felt fairly intuitive
  - (to a C++ programmer)

Nifty Little Features

- D has many nifty features that make the language pleasant to use
- Not revolutionary, but common sense
- Many small features were a pleasant surprise

foreach

```
foreach (value; iterable)
   doSomething(value);


foreach (key, value; iterable)
   doSomething(key, value);


foreach (regNo, localIdx; gpRegMap)
{
    if (localIdx is NULL_LOCAL)
        continue;

    spillReg(as, regNo);
}
```

in and !in

```
key in map
```

```
(key in map) == false
```

```
key !in map
```

```
// Collect the dead functions
foreach (ptr, fun; interp.funRefs)
    if (ptr !in interp.liveFuns)
        collectFun(interp, fun);
```

Type Inference

```
auto interp = new Interp();


auto getExportAddr(string name)
{
    assert (
        name in this.exports,
        "invalid exported label"
    );

    return getAddress(this.exports[name]);
}
```

Delegates

```
// mov
test(
    delegate void (Assembler a) { a.instr(MOV, EAX, 7); },
    "B807000000"
);
test(
    delegate void (Assembler a) { a.instr(MOV, EAX, EBX); },
    "89D8"
);
```

Type Ranges

```
size_t immSize() const
{
    // Compute the smallest size this immediate fits in
    if (imm >= int8_t.min && imm <= int8_t.max)
        return 8;
    if (imm >= int16_t.min && imm <= int16_t.max)
        return 16;
    if (imm >= int32_t.min && imm <= int32_t.max)
        return 32;

    return 64;
}
```

The Garbage Collector

- Had to make the Higgs and D GCs work together
  - Manual memory allocation
  - Regions of memory not collected by D
  - Maintain references to D heap alive
- Worked better than expected
  - D GC behaves predictably
  - Haven't had many bugs

Templates + Mixins

```
extern (C) void ArithOp(Type typeTag, uint arity, string op)
(Interp interp, IRInstr instr)

alias ArithOp!(Type.INT32, 2, "auto r = x + y;") op_add_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x - y;") op_sub_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x * y;") op_mul_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x / y;") op_div_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x % y;") op_mod_i32;

alias ArithOp!(Type.INT32, 2, "auto r = x & y;") op_and_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x | y;") op_or_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x ^ y;") op_xor_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x << y;") op_lsft_i32;
alias ArithOp!(Type.INT32, 2, "auto r = x >> y;") op_rsft_i32;
```

The Build System

- Faster build times than other languages
- Much simpler than C/C++ makefiles:
  - Pass source files to the compiler
  - Things get compiled
  - You are done
- Reduces need for complex build tools
  - Higgs uses one short makefile

The Community

- Centralized dlang.org website
  - Forums, documentation, downloads
- Responsive, enthusiastic community
  - Received answers to all my questions
- Most languages don't have a go-to place
  - Many isolated resources

Compile-Time Function Evaluation

- One of the reasons I chose D is CTFE
- Mixins: powerful macro system
  - Allows creating domain-specific languages
  - Arguably D's most powerful feature
- Unfortunately, ran into issues

Declarative Object Layouts

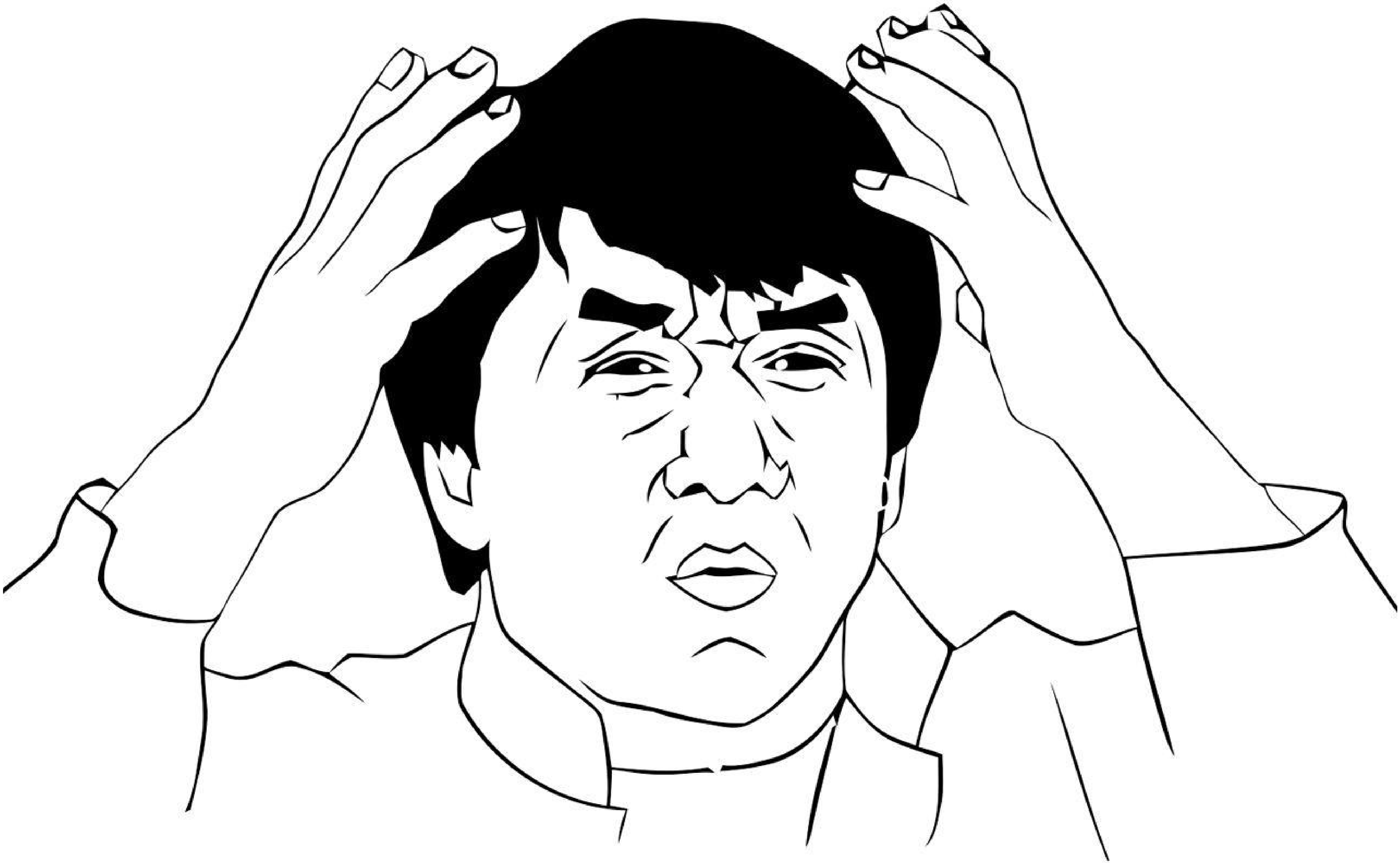
- Want to control memory layout of our own objects precisely
- Access to objects from both D and JS
- Layouts described in declarative form
- D and JS code for getters/setters, allocation, initialization and GC traversal is auto-generated at compile-time
- Make domain-specific language using mixins

```
mixin(
genLayouts([

    // String layout
    Layout(
        "str",
        null,
        [
            Field("len" , "uint32"),        // String length
            Field("hash", "uint32"),        // Hash code
            Field("data", "uint16", "len") // UTF-16 character data
        ]
    ),

    // String table layout (for hash consing)
    Layout(
        "strtbl",
        null,
        [
            Field("cap" , "uint32"),                // Capacity
            Field("num_strs" , "uint32", "", "0"), // Number of strings
            Field("str", "refptr", "cap", "null"), // Array of strings
        ]
    ),

    …
]));
```

CTFE broke down

- Generating a few thousand lines of source code became very slow
- Memory leak using all available memory
- Computer locked up during compilation

“This problem is well known [...] but **it will take time to fix it well, possibly some months or more**.”



```
import std.string;
import std.array;
import std.conv;

string fun()
{
    auto app = appender!string();

    for (size_t i = 0; i < 10000; ++i)
        app.put("const int x ~" ~ to!string(i) ~ " = 0;");

    return app.data;
}

mixin(fun());
```

Template Issues

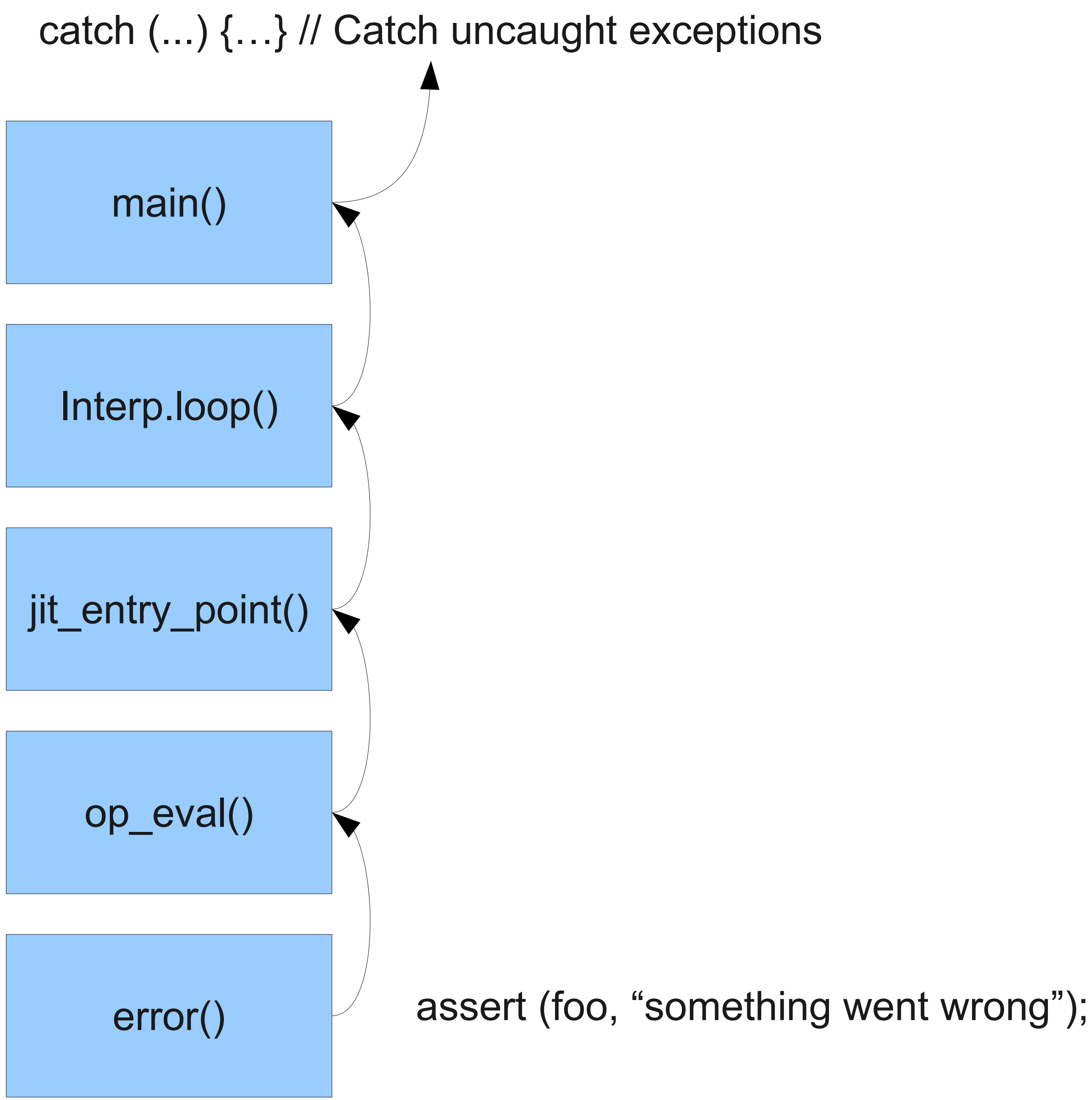
- Needed template with list of integer arguments
- Known compiler bug
- Had to accept code duplication

```
mixin template MyTemplate(int[] arr) {}
```

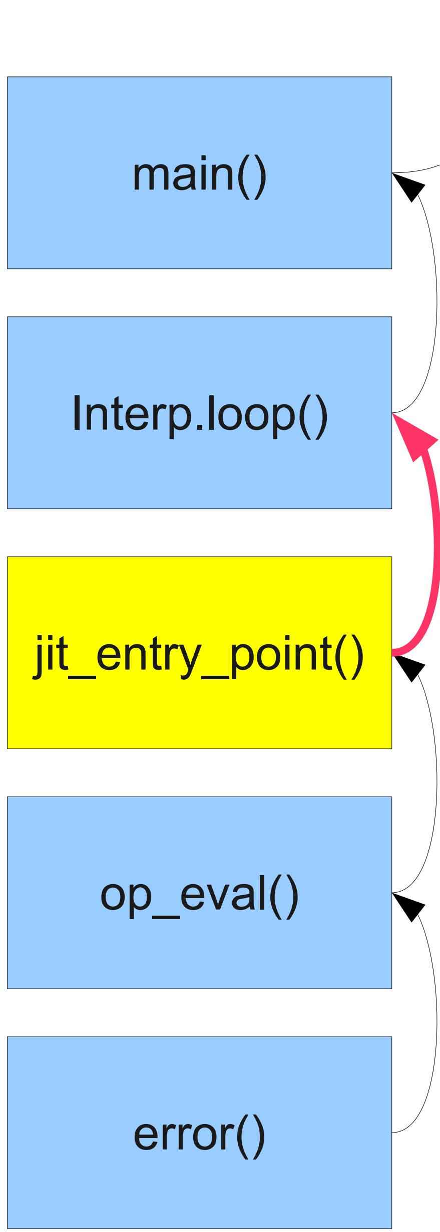
```
Error: arithmetic/string type expected for value-
parameter, not int[]
```

The `assert` that segfaults

- Tripped assert causes segfault when in a function indirectly called by generated code
- Tries to unwind the stack and fails
- `assert` meant to provide useful info if something goes wrong
- Should probably print an error before attempting to unwind the stack


catch (...) {…} // Catch uncaught exceptions
main()
Interp.loop()
jit_entry_point()
op_eval()
error()
assert (foo, “something went wrong”);

catch (...) {…} // Catch uncaught exceptions



One of these frames is not like the others,
one of these frames just doesn't belong!

assert (foo, “something went wrong”);

Unit Tests Blocks

- Don't support naming unit tests
- Failing tests not reported at the end
- The `main` function is still called normally
  - Higgs starts a REPL by default
- No way to select which tests are run
- Tempted to write our own framework

```
alias void function(CodeGenCtx ctx, CodeGenState st,
IRInstr instr) CodeGenFn;

CodeGenFn[Opcode*] codeGenFns;

/// Map opcodes to JIT code generation functions
static this()
{
    codeGenFns[&SET_TRUE]        = &gen_set_true;
    codeGenFns[&SET_FALSE]       = &gen_set_false;
    codeGenFns[&SET_UNDEF]       = &gen_set_undef;
    codeGenFns[&SET_MISSING]     = &gen_set_missing;
    codeGenFns[&SET_NULL]        = &gen_set_null;
    codeGenFns[&SET_INT32]       = &gen_set_int32;
    codeGenFns[&SET_STR]         = &gen_set_str;

    codeGenFns[&MOVE]            = &gen_move;

    codeGenFns[&IS_CONST]        = &gen_is_const;
    codeGenFns[&IS_REFPTR]       = &gen_is_refptr;
    codeGenFns[&IS_INT32]        = &gen_is_int32;
    codeGenFns[&IS_FLOAT]        = &gen_is_float;

   ...
}
```

A JIT for D's CTFE?

The Cost of JIT

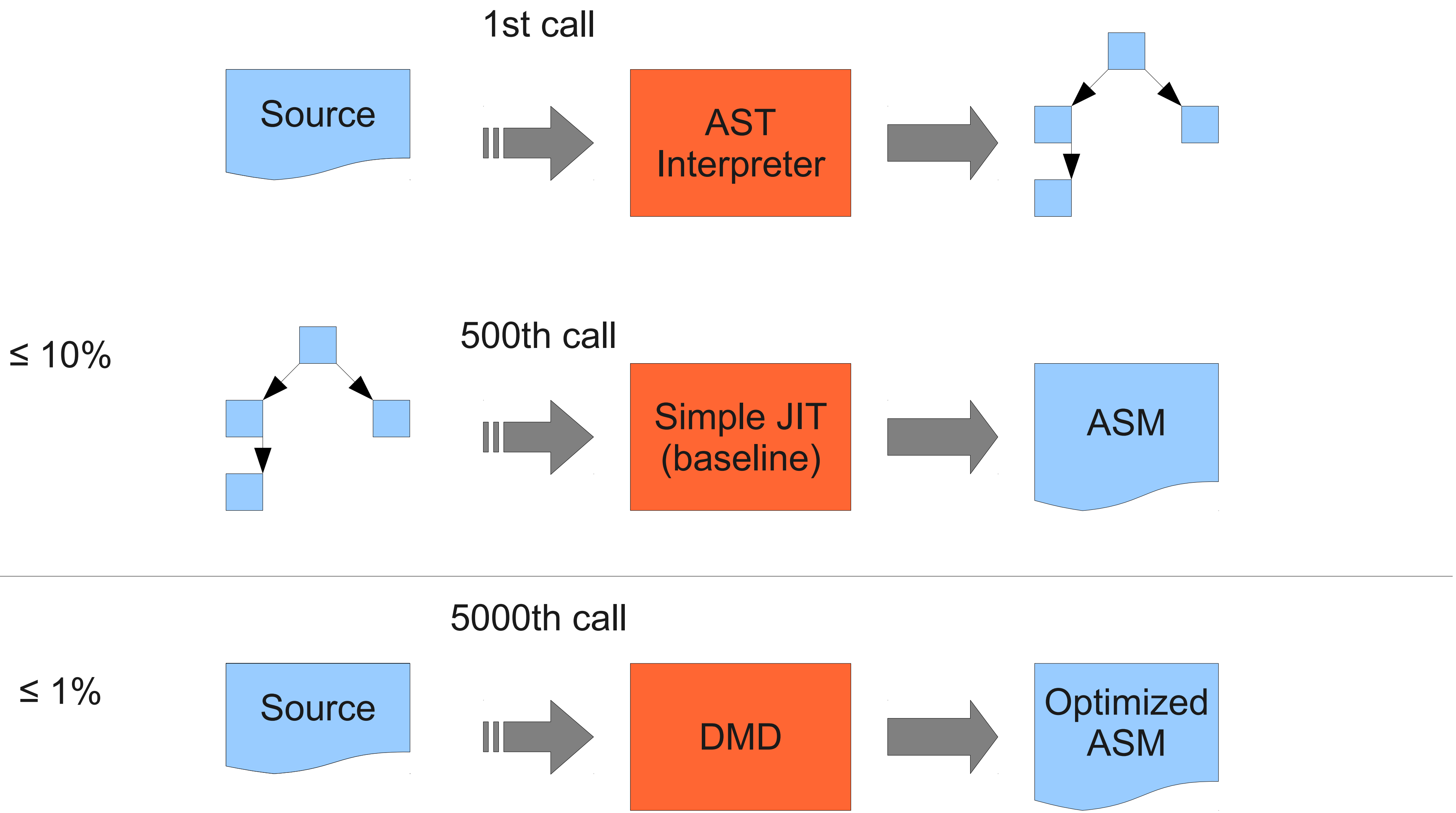
- Mainstream VMs typically have a JIT with multiple optimization levels
  - Or an interpreter and a JIT (e.g.: Firefox, Higgs)
- JIT compilation takes time, must pay for itself
  - Not worth it for functions that only run a few times
  - Only worthwhile for heavier computational loads
- Majority of code never gets optimized
  - Doesn't run for very long, if at all

Does CTFE need a JIT?

- What kinds of things are people doing with it?
  - Typical scenario: source generation for mixin
  - At most a few thousand string concatenations
  - Probably don't need fast CTFE for this
- Be open minded: faster CTFE opens doors
  - Generating procedural content at compile time
  - “If you build it, they will come”

A Simple Architecture

- Don't bother optimizing the interpreter
  - Mozilla is planning to switch to an AST interpreter
- Start with a simple JIT
  - e.g.: stack-based, no register allocation
  - Will compile very fast
  - Will be much faster than your interpreter
- Reuse some of the D compilation infrastructure?
  - Compile the really hot code with DMD
  - Reuse compiled code between CTFE runs


1st call
Source
AST Interpreter
≤ 10%
500th call
Simple JIT (baseline)
ASM
5000th call
≤ 1%
Source
DMD
Optimized ASM

Other Considerations

- Precompile most library code used in CTFE
  - Interpreter can call into compiled code
  - i.e.: most string/array operations
  - Some templates can be precompiled
- Re-optimizing mid-call complicates things
  - Long-running functions
  - Probably not a concern

Suggestions

Static Initialization of Maps

- Associative arrays are useful for declarative programming
- Can't currently statically initialize them in D
  - Requires using static constructors
- Is possible in JS, dynamic languages
- Would be helpful if this feature was in D
  - Still useful if limited to constant maps

Integer Types

- D integer types have guaranteed sizes, but they're not obvious from the name
- Why not have int8, uint8, int32, uint32, etc. in default namespace, encourage their use?
- Make programmers more aware of the limitations/characteristics of the type they're using.

Documentation Effort

- Expose people to more idiomatic code
- dlang.org, Documentation->Articles
  - Few things in there
  - Most not that useful for beginners
- Expand/promote tutorials
  - Show people the cool things you can do with D

Conclusion

- Overall positive experience using D
- Some hiccups, but no showstoppers
  - Unexpected use cases
- People accuse C++ of being too complex
  - D has all the features, feels like cohesive whole
  - Re-engineered with hindsight
- More productive than writing C++

github.com/maximecb/Higgs

maximechevalierb@gmail.com

pointersgonewild.wordpress.com

Love2Code on twitter

Special Thanks To

- Thesis advisors: Bruno Dufour, Marc Feeley
- Contributors: Tom Brasington, John Colvin
- Supporters: Erinn
- The Mozilla Foundation
- Andrei Alexandrescu and Walter Bright
- The flying spaghetti monster