Higgs, an Experimental JIT Compiler written in D

DConf 2013

Maxime Chevalier-Boisvert
Université de Montréal

Introduction

- PhD research: compilers, dynamic language optimization, type analysis
- 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 experience implementing a JIT in D
  - Implementation of a JIT for D's CTFE

Dynamic Languages

- Typically have
  - No type annotations
  - Late binding
  - Dynamic typing
  - Dynamic loading of code (eval, load)
  - Dynamic growth of objects
- Are not
  - “Untyped” languages
  - “Interpreted” languages

Why so Slow?

- Naive implementations have significant overhead
- Values are usually “boxed”
  - Values as pairs: datum + type
  - CPython's numbers are objects!
- Every operator (+, -, *, etc.) has dynamic dispatch
- Global accesses and method calls require expensive hash table lookups
- Easiest to implement in an interpreter

Optimization

- Optimizing dynamic languages is largely about removing unnecessary overhead
- Tools:
  - Guessing likely types with heuristics
  - Measuring likely types with profiling
  - Proving types using type inference
- Goals:
  - Unboxing values
  - Removing dispatch overhead
  - Inlining function calls

Harder than it seems

- JS, Python, Ruby not designed with optimization in mind
- Dynamic code loading, eval
  - New code could be loaded at any point
  - New code can break your assumptions
- Numerical towers, overflow checks
  - Hard to prove overflows won't happen

The Future

- Efficiently compiling dynamic languages requires type information
- Translating programs into more static code
- Type analysis, type inference
- Prove that specific variables have a given type
  - e.g.: x is always an integer
  - e.g.: the function foo will never be redefined

JIT Compilers

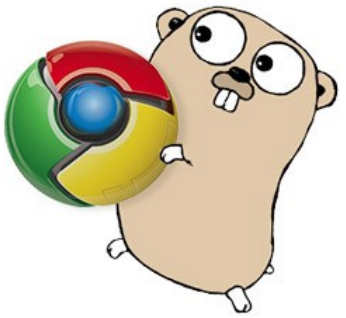
- Need access to low-level operations
  - Manual memory management
  - Raw memory access
  - System libraries
- Are very complex pieces of software
  - Pipeline of code transformations
  - Several interacting components
- Wish for
  - Expressive language
  - Garbage collection

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
  - Boehm's GC: it will *probably* work™

Other Options

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



- Mozilla's Rust
  - Very young, still changing, in flux
  - Not a realistic 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

Higgs

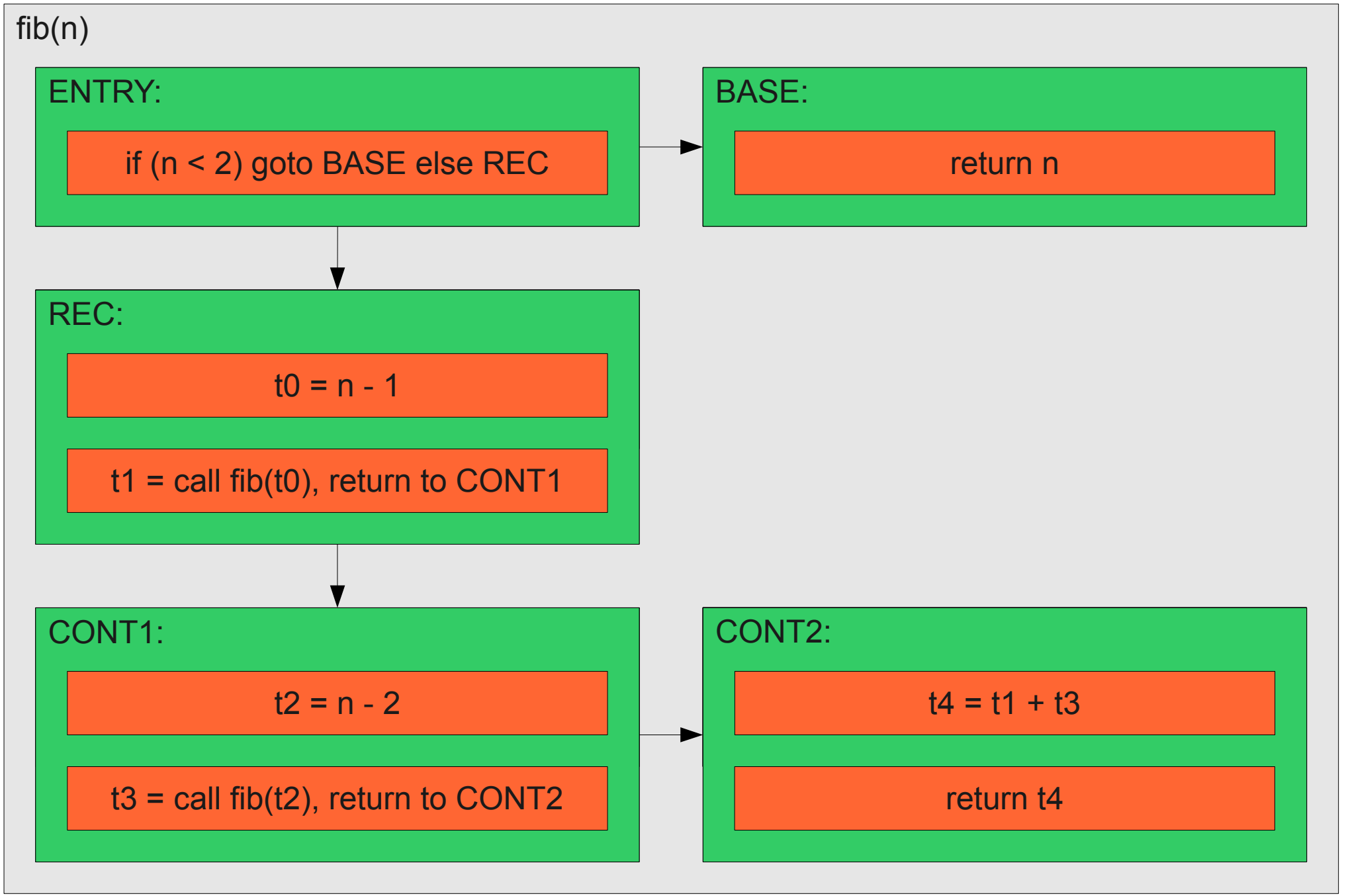
- Two main components:
  - Interpreter
  - JIT compiler
- Complexity:
  - D: ~23 KLOC
  - JS: ~11 KLOC
  - Python: ~2 KLOC
- JS support:
  - ~ES5, no property attributes, no `with`

The Interpreter

- Interpreter is used:
  - For profiling
  - As a "fallback" for unimplemented JIT features
  - To get started quickly. JIT compilation has a cost
- Designed to be:
  - Simple, easy to maintain
  - Quick to extend and experiment with
  - "JIT-friendly"
- Interpreter is quite slow, 1000 cycles/instr

JIT-Friendly

- Register based, not stack-based
  - Easier to analyze/optimize
- IR is based on a control-flow graph, not an AST
  - More amenable to JIT compilation
  - Simpler, easier to reason about (IP, targets)
- Interpreter stack is an array of values
  - Directly reused by the JIT


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

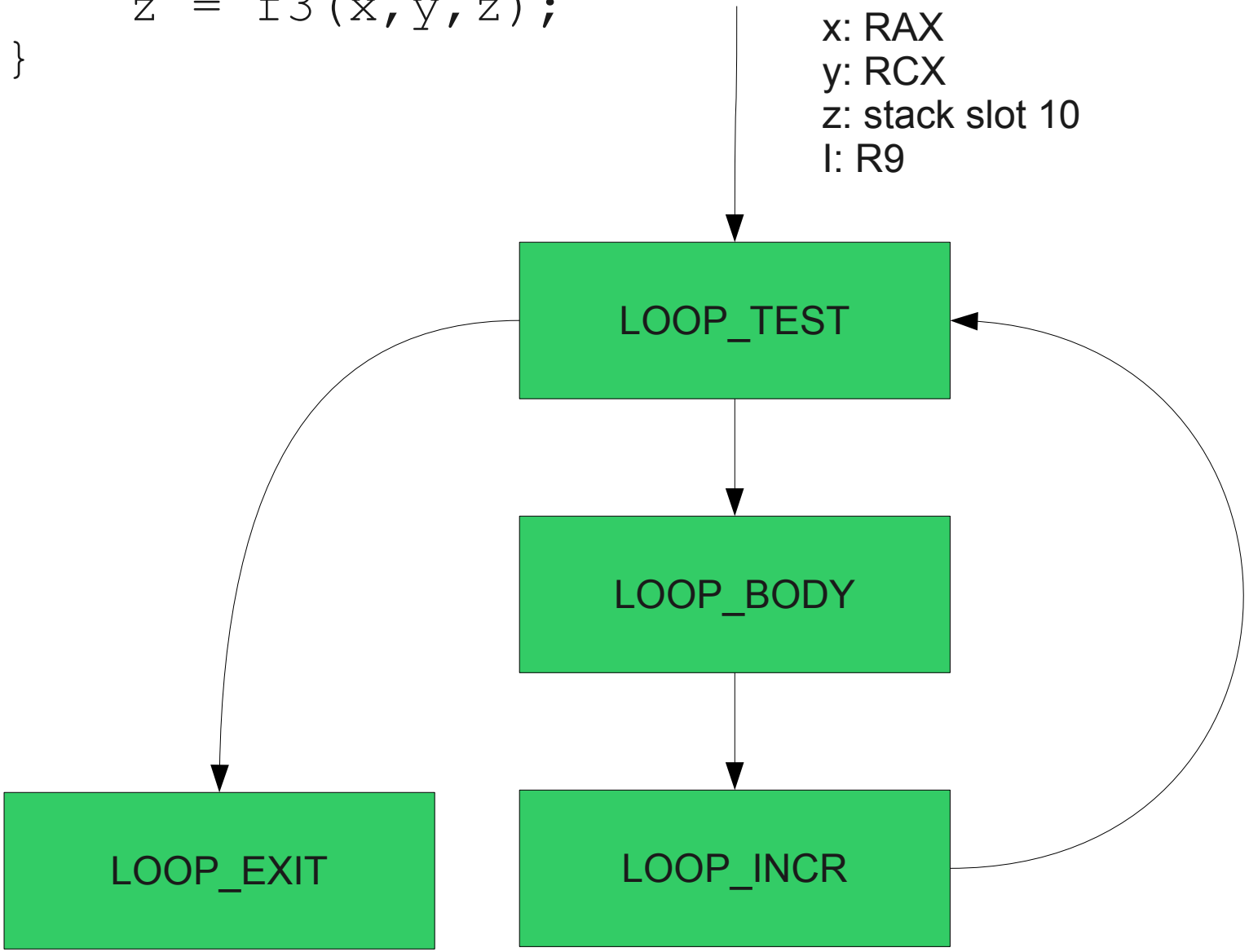
The JIT Compiler

- Targets x86-64 only, for simplicity. Easier than supporting just x86 32 bits.
- Kicks in relatively quickly, after functions have been found hot enough (worth compiling)
  - Execution counters on basic blocks
- Speedups of 5 to 20x
  - Expected to soon reach 100x+ speedups
- Currently fairly basic
  - No inlining, bulk of the code is function calls

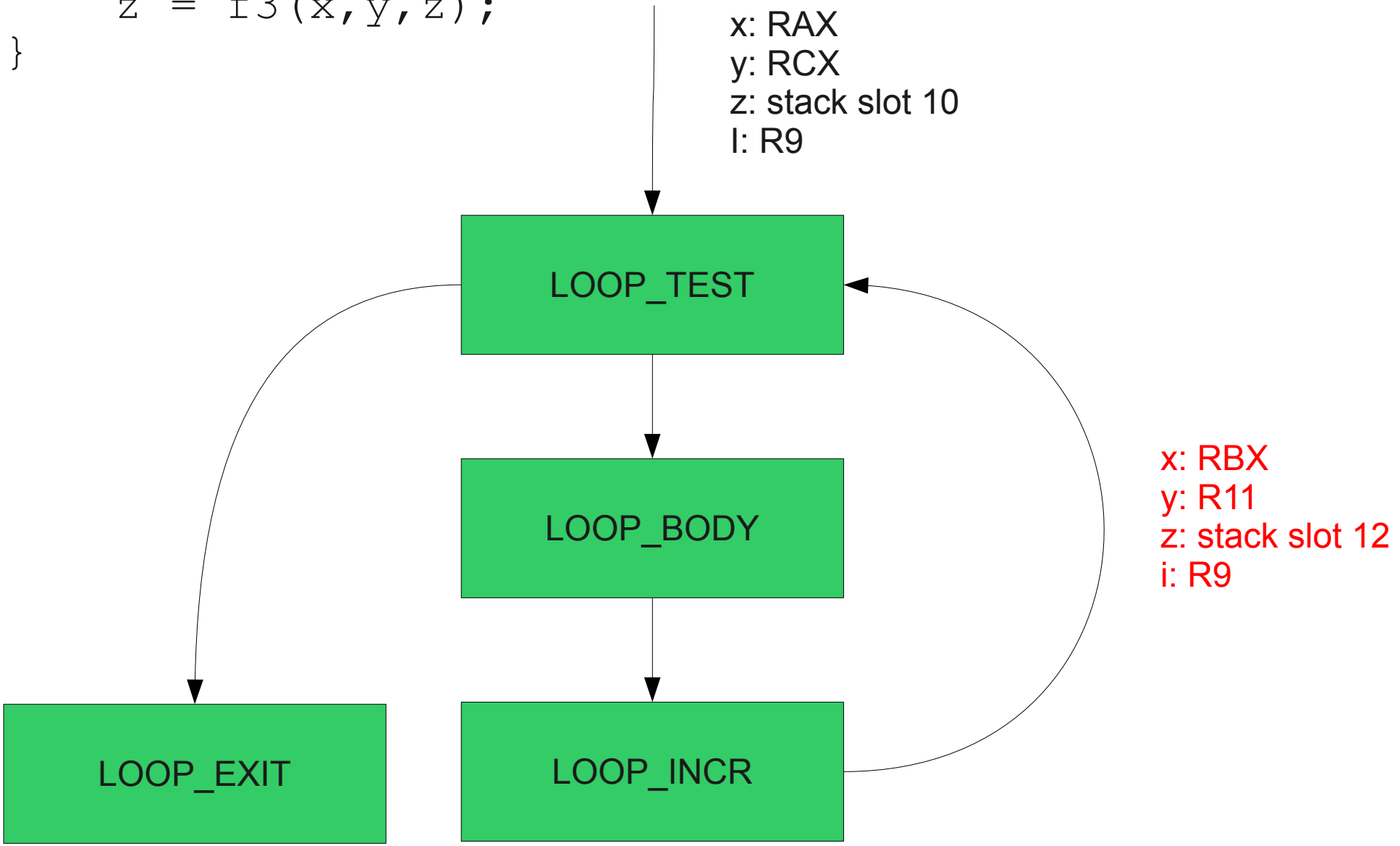
Current Research

- Context-driven basic block versioning (cloning)
- Context is:
  - Low-level type information
  - Register allocation state
  - Accumulated facts
- Currently integrating this in the backend (JIT)
- Similarities with trace compilation

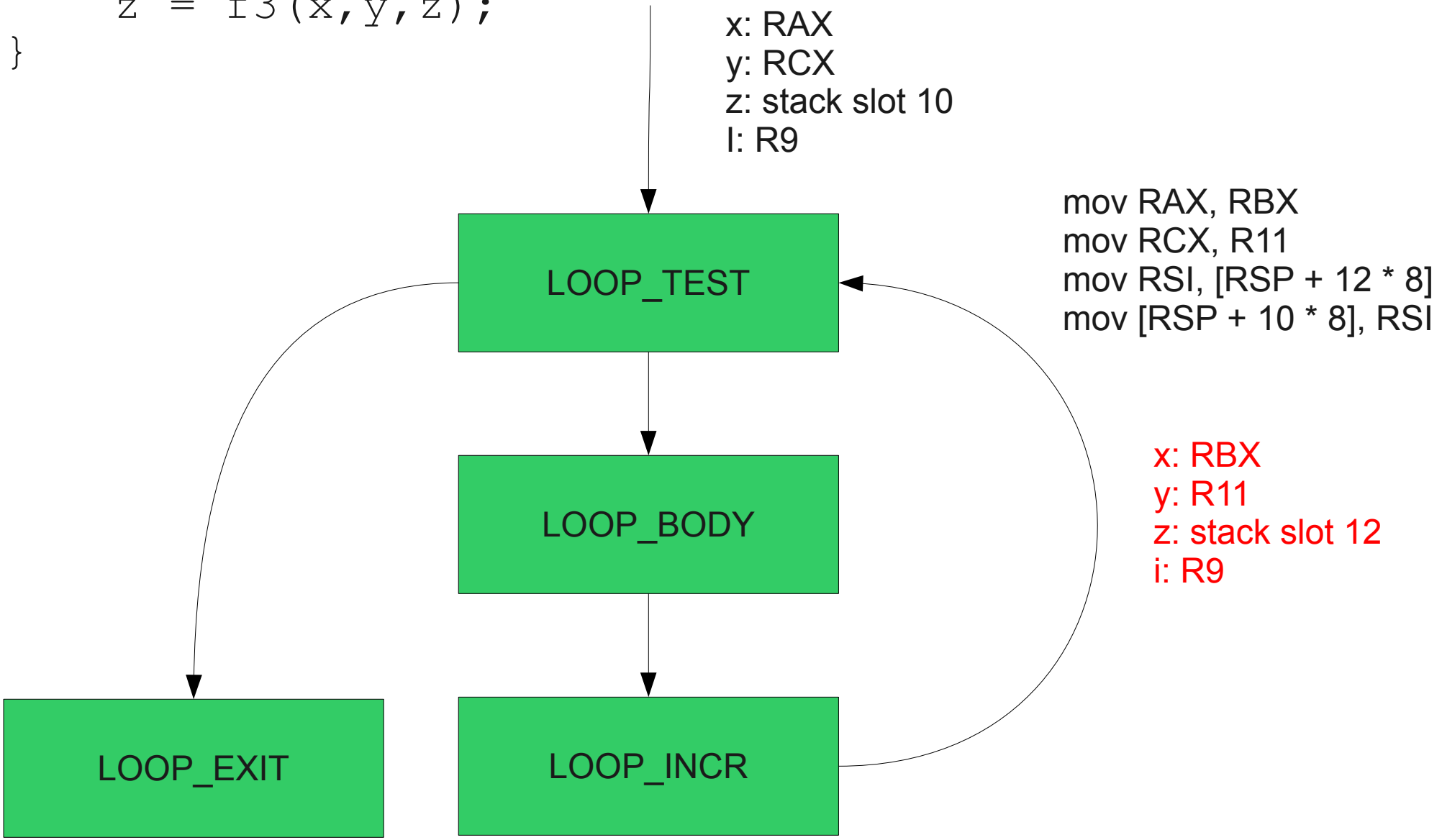
```
for (i = 0; i < k; ++i) {
    x = f1(x,y,z);
    y = f2(x,y,z);
    z = f3(x,y,z);
}
```

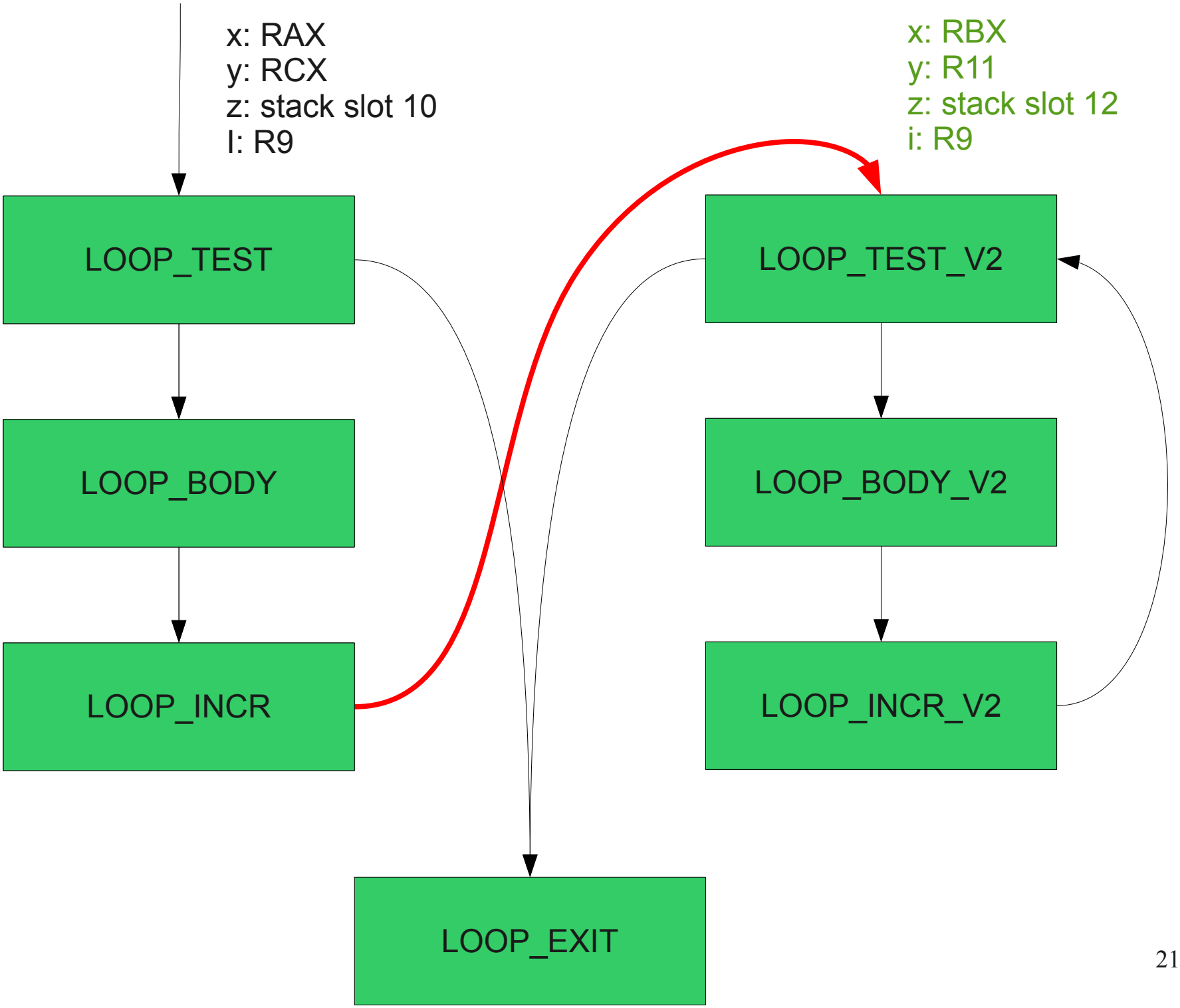


```
for (i = 0; i < k; ++i) {
    x = f1(x,y,z);
    y = f2(x,y,z);
    z = f3(x,y,z);
}
```



```
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

A “Multi-world” View

- Traditional control-flow analysis
  - Compute a fixed-point (LFP or GFP)
  - At each basic block, solution must agree
  - Find pessimistic answer that agrees with all inputs
- Block versioning
  - Multiple solutions possible for a block
  - All possible answers can be simultaneously true
  - Don't necessarily have to sacrifice
  - Fixed point on the creation of new blocks

Advantages

- Automatically do loop peeling (when useful)
- Automatically do tail duplication
- May do some amount of loop unrolling
- Similar to trace compilation
  - Accumulate knowledge (e.g.: from type tests)
  - Specialize code based on types
  - Specialize based on constants
- Register allocation
  - Fewer move operations
  - Make simpler allocators more efficient

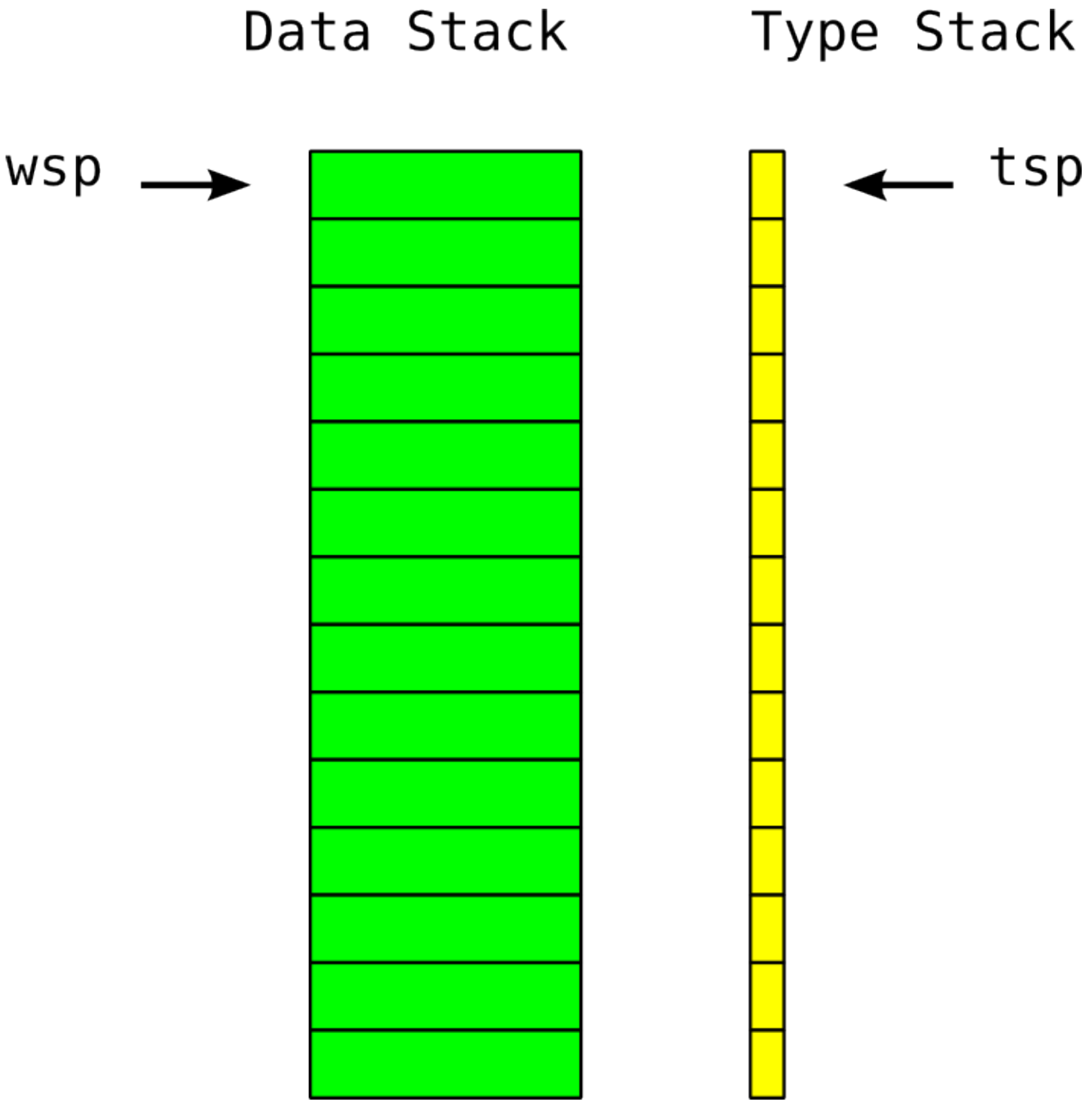
Research Questions

- 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?

Design Choices

Split Tagging

- Higgs uses a split tagging scheme
  - No tag bits, no NaN tagging
  - One value word (64-bit) + one type tag byte
- Downside: size, two stack pointers, two arrays
- Upsides:
  - Values accessible directly, no unboxing
  - Modern CPUs have multiple execution units
  - In many cases, can completely ignore type info
  - JIT performance favored over interpreter performance


Data Stack
Type Stack
wsp
tsp

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!

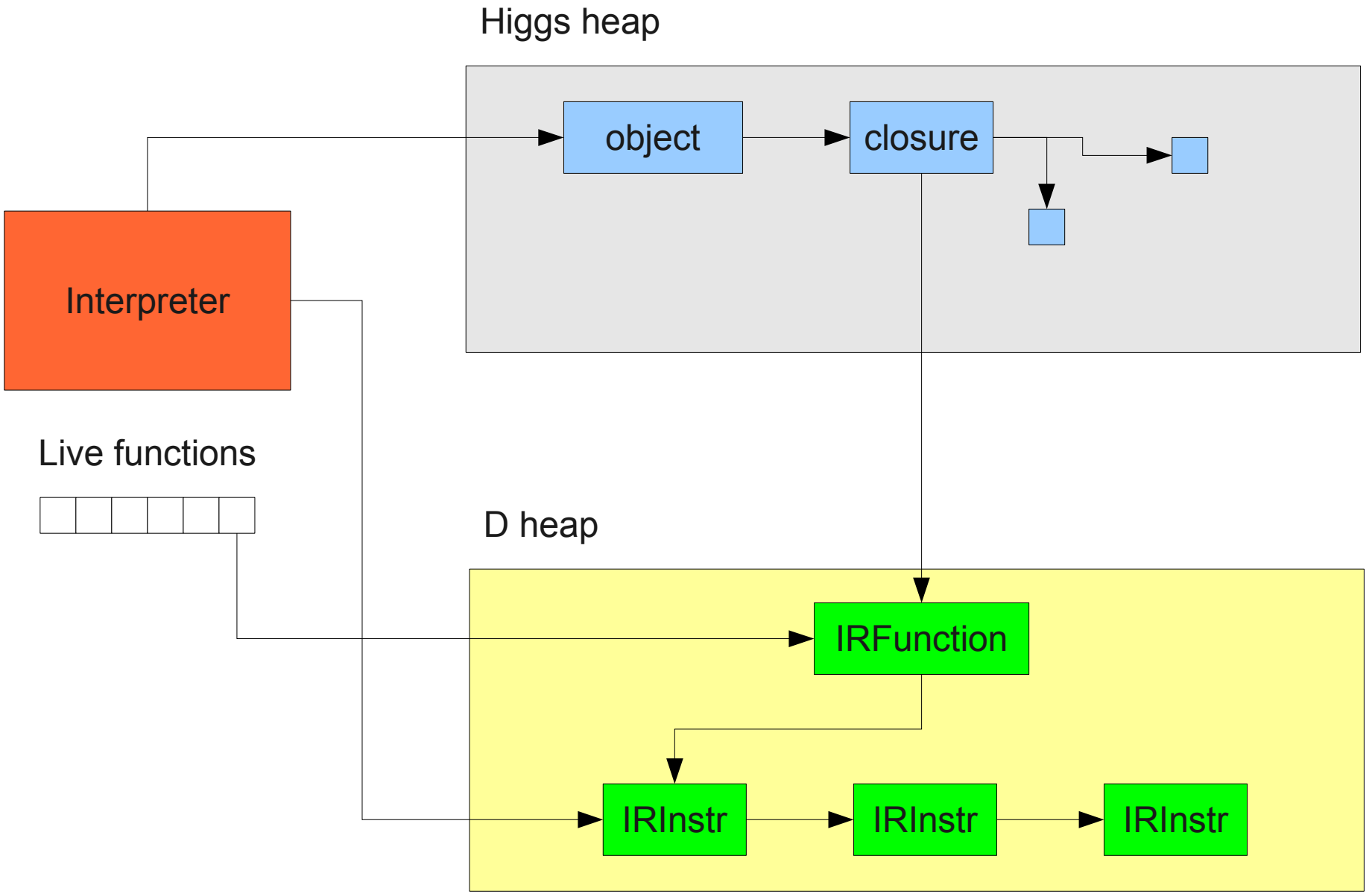
Self-hosting

- 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 one pointer
  - Collection time proportional to live data
- References to D objects need to 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

Declarative Object Layouts

- Want to control memory layout of our own objects precisely
- Want access to objects from both D and JS
- Object layouts described in declarative form
- D and JS code for getters/setters, allocation, initialization and GC traversal is auto-generated at compile-time

Interpreter Fallback

- JIT uses interpreter stack for spills
- JIT avoids compiling unexecuted blocks.
- Fast paths for common/expected cases
  - Some slow paths use interpreter
- Some IR instructions handled by interpreter
  - Allows incremental JIT construction/refactoring

What Worked Well

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
  - D has better/simpler ways of doing certain things
- Felt fairly intuitive
  - (to a C++ programmer)

Nifty Little Features

- D has many nifty little 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);
```

auto

```
auto interp = new Interp();


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

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

String Concatenation

```
return this.name ~ "(" ~ idString() ~ ")";
```

```
output ~= "(" ~ lStr ~ ")";
```

```
output ~= to!string(arg.int32Val);
```

delegates

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

Casting + declaration + test

```
// Function (closure) as an expression
if (auto funExpr = cast(FunExpr)expr)
{
    // Resolve variable declarations and
    // references in the nested function
    resolveVars(funExpr, s);
}
```

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

Templated Instructions

```
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;
```

CTFE in Templates

```
/// Increment a global JIT stat counter variable
void incStatCnt(string varName)(Assembler as)
{
    if (!opts.jit_stats)
        return;

    mixin("auto vSize = " ~ varName ~ ".sizeof;");
    mixin("auto vAddr = &" ~ varName ~ ";");

    as.ptr(RAX, vAddr);

    as.instr(INC, new X86Mem(vSize * 8, RAX));
}
```

The build system

- Faster build times than other static 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

The Community

- Centralized dlang.org website
- Public web forums
- Responsive, enthusiastic community
- Most languages don't have this

Problems Encountered

Compile-Time Function Evaluation

- One of the reasons I chose D is CTFE
- Allows generating code at compile-time
- Powerful macro system
  - Much more elegant than C's
  - Allows creating domain-specific languages
- Hyped, novel feature of the D language
  - Argualbly D's most powerful feature

CTFE's quirks

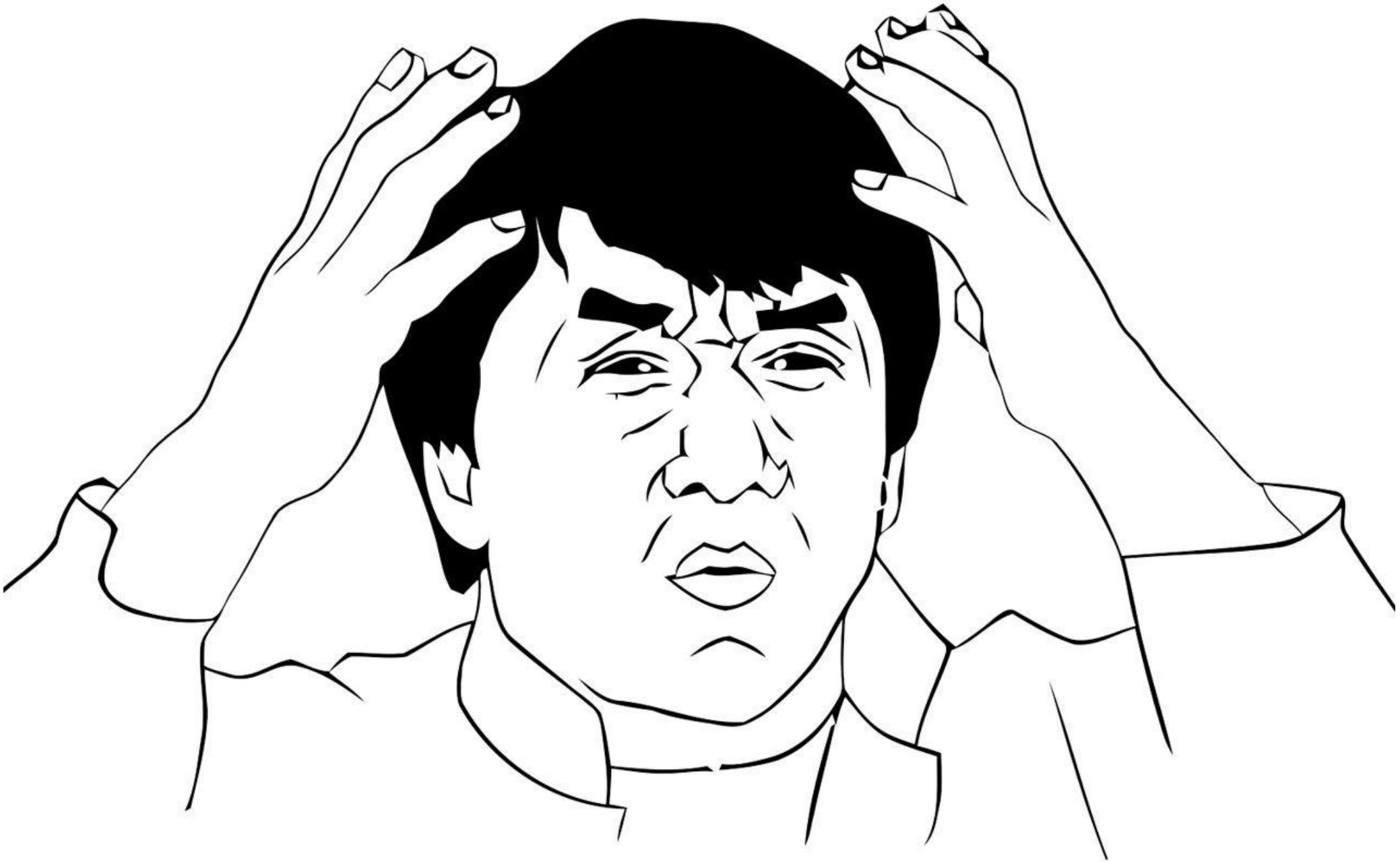
- One of the main uses of CTFE is to generate strings for use in mixin blocks
  - At the time, no support for format (*now fixed*)
- Wish to use writef statements for debugging
  - But these are not supported
  - Told to use impractical pragma to print output

```
/usr/include/dmd/phobos/std/stdio.d(1614): Error:
fprintf cannot be interpreted at compile time, because
it has no available source code
```

CTFE broke down

- Generating a few thousand lines of source code became very slow
- Computer locked up during compilation
  - Memory leak ended up using all available memory
- Told this issue would be fixed in several months
  - Could not wait several months
- Ended up rewriting declarative source generation code in Python

“There are coding strategies to partially reduce the memory used during CTFE, but in general it uses lot of memory, sometimes too much. 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

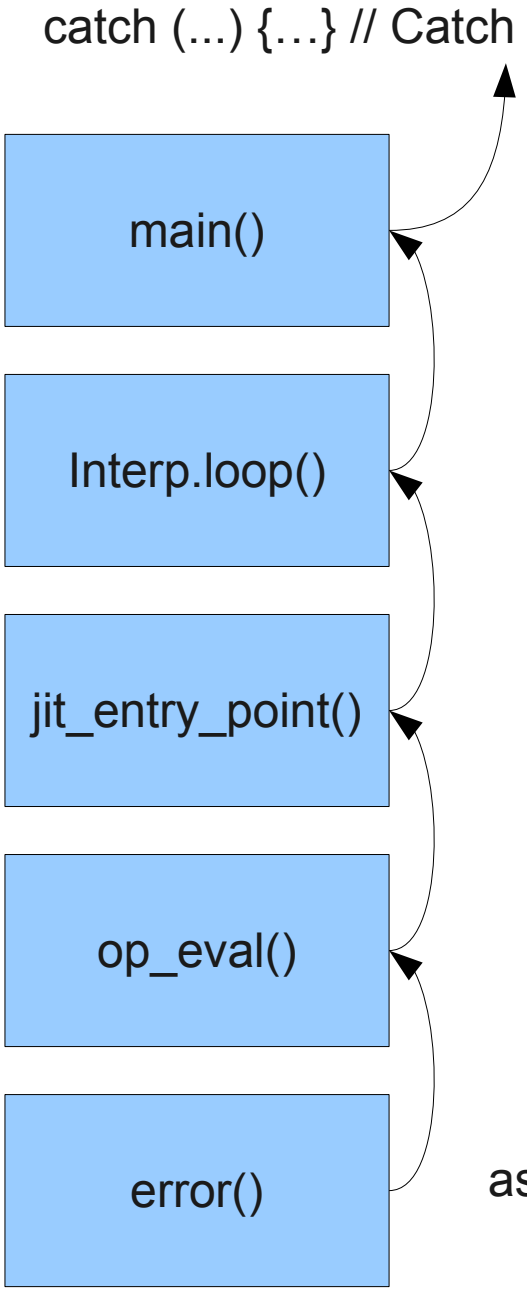
- Needed template with list of integer arguments
- Known compiler bug
- Had to accept code duplication

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

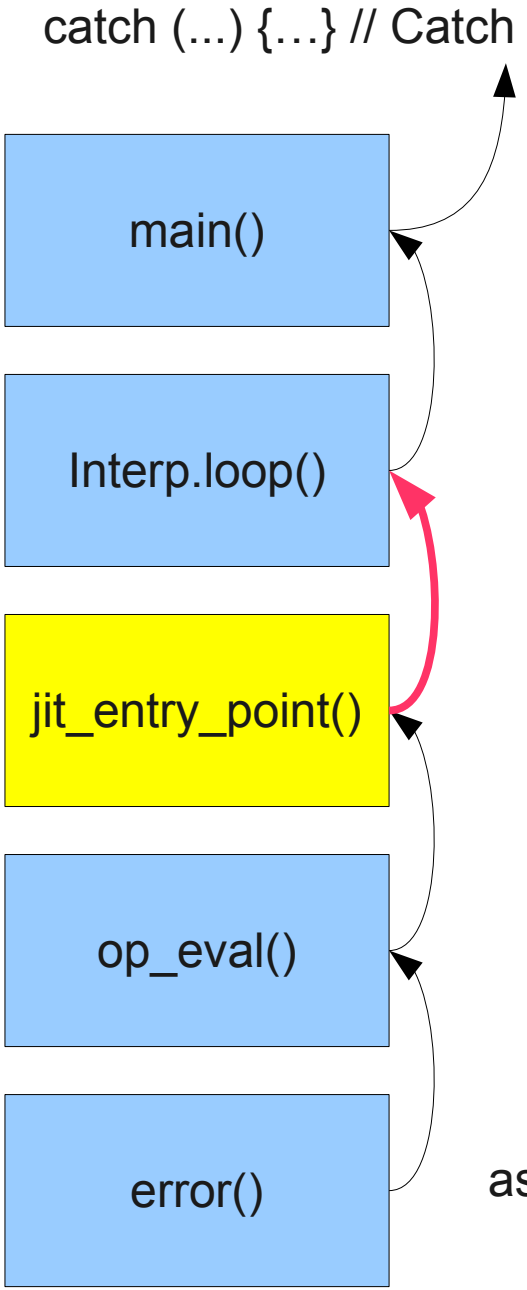
```
Error: arithmetic/string type expected for value-
parameter, not int[]
```

The `assert` that segfaults

- Tripped assert causes segfault when in a function indirectly called by generated code
- Segfaults, prints nothing
- Tries to unwind the stack and fails
- `assert` statements are meant to provide useful info if something goes wrong
- Should probably print an error before attempting to unwind the stack



assert (foo, “something went wrong”);



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

assert (foo, “something went wrong”);

D's Unit Tests Support

- Doesn't support naming unit tests
- Doesn't log passing or failing unit tests
- Failing tests not reported at the end
- The `main` function is still called normally
  - Higgs starts a REPL by default
- Token support for unit tests
- 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;

    ...
}
```

Cyclic Module Dependencies

- Aborting: Cycle detected between modules with ctors/dtors
- Why does this happen?
  - Prevent module A accessing unininitialized module B?
- Possibly too strict
  - False dependencies most of the time
- Forces division of code into additional modules based on compiler limitations
  - Somewhat reminiscent of C++ header issues

A JIT for D's CTFE?

The Cost of JIT

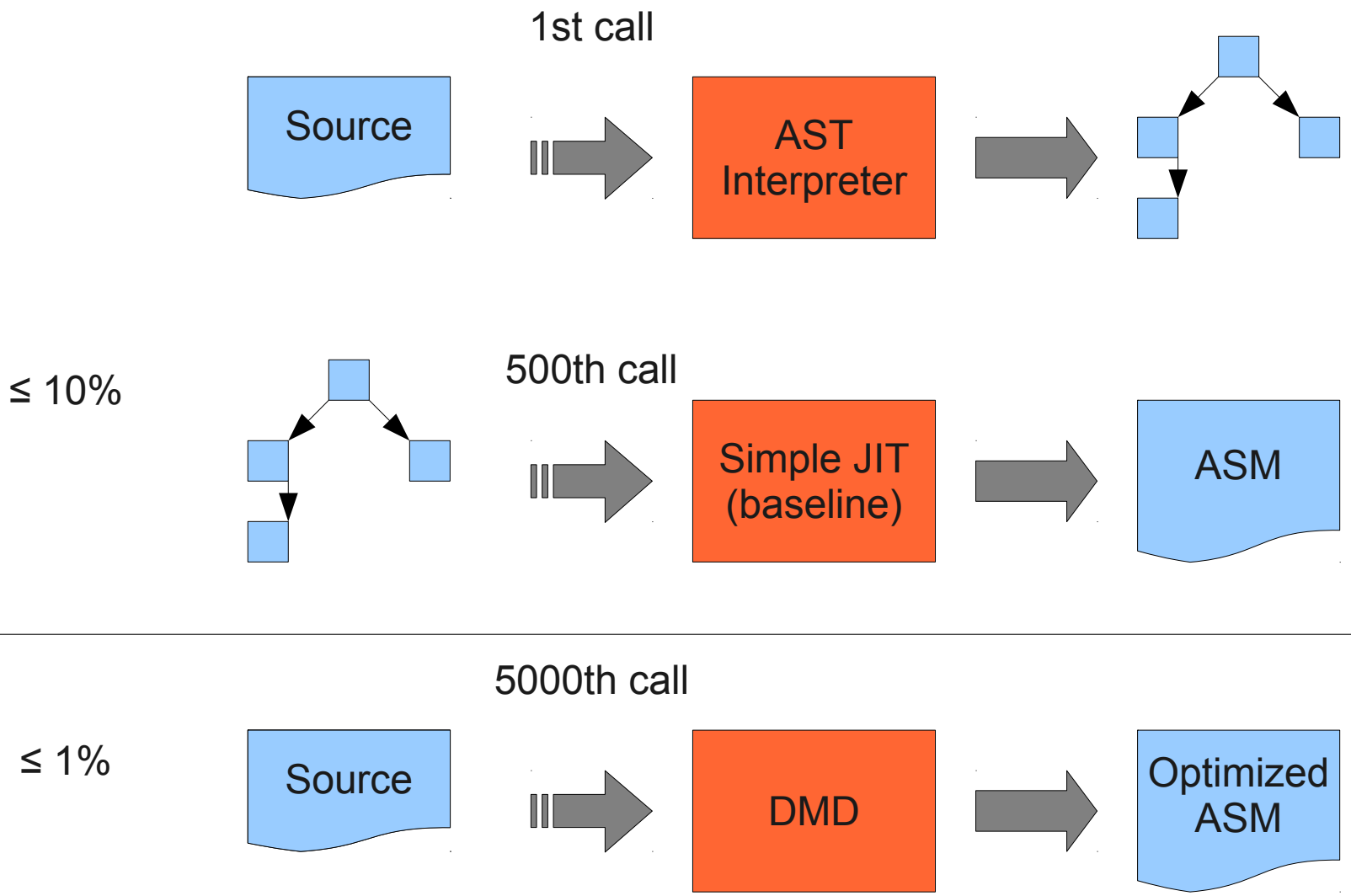
- Mainstream VMs typically have a JIT with multiple optimization levels
  - Or an interpreter and a JIT (e.g.: Firefox, Higgs)
- JIT compilation takes time, must pay for itself
  - Not worth it for functions that only run a few times
  - Only worthwhile for heavier computational loads
- Majority of code 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

Bug Fixing Effort

- Fixing CTFE bugs is critical
- Bugs like these can scare people away
  - Need to be fixed very rapidly
- Make bugtracker more visible
- Allow users to vote on bug fixing priority
  - Some bugs will affect many people

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

- Make the language more accessible
- Expose people to more idiomatic code
- dlang.org, Documentation->Articles
  - Few things in there, most not that useful for beginners
  - Articles/Tutorials should be the first thing under Documentation
- Expand/promote tutorials
  - Show people the cool things you can do with D

Conclusion

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

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 flying spaghetti monster