Nim - the first high performance language with full support for hot code-reloading at runtime

by Viktor Kirilov

Me, myself and I

- my name is Viktor Kirilov - from Bulgaria
- creator of doctest - the **fastest** C++ testing framework
- apparently I like text-heavy slides and reading from them...!
  - deal with it : |

Talk agenda

- some Nim code
- the performant programming language landscape
  - read: heavily biased C++ rant
- Nim compilation model
- hot code reloading
  - usage & implementation
  - ".dll" => assume .so/.dylib (platform-agnostic)
- demo
- comments & conclusions
- a bit on REPLs

Hello

```
1  echo "Hello World"
```

Currencies

```nim
type
  # or use {.borrow.} here to inherit everything
  Dollars* = distinct float

proc `+` *(a, b: Dollars): Dollars {.borrow.}

var a = 20.Dollars

a = 25  # Doesn't compile
a = 25.Dollars  # Works fine

a = 20.Dollars * 20.Dollars # Doesn't compile
a = 20.Dollars + 20.Dollars # Works fine
```

Sets

Operator	Description	Example Code
`a in B`	is a an element of B?	`'d' in {'a'..'z'}`
`a notin B`	is a not an element of B?	`40 notin {2..20}`
`A + B`	union of A with B	`{'a'..'m'} + {'n'..'z'} == {'a'..'z'}`
`A - B`	relative complement of A in B	`{'a'..'z'} - {'b'..'d'} == {'a', 'e'..'z'}`
`A + {b}`	add element b to set A	`{'b'..'z'} + {'a'} == {'a'..'z'}`
`A - {b}`	remove element b from set A	`{'a'..'z'} - {'a'} == {'b'..'z'}`
`A * B`	intersection of A with B	`{'a'..'m'} * {'c'..'z'} == {'c'..'m'}`
`A <= B`	is A a subset of B?	`{'a'..'c'} <= {'a'..'z'}`
`A < B`	is A a strict subset of B?	`{'b'..'c'} < {'a'..'z'}`

Iterators

```
type
  CustomRange = object
    low: int
    high: int

iterator items(range: CustomRange): int =
  var i = range.low
  while i <= range.high:
    yield i
    inc i

iterator pairs(range: CustomRange): tuple[a: int, b: char] =
  for i in range:  # uses CustomRange.items
    yield (i, char(i + ord('a')))

for i, c in CustomRange(low: 1, high: 3):
  echo c

# prints: b, c, d
```

Variants

```nim
# This is an example how an abstract syntax tree could be modelled in Nim
type
  NodeKind = enum  # the different node types
    nkInt,          # a leaf with an integer value
    nkFloat,        # a leaf with a float value
    nkString,       # a leaf with a string value
    nkAdd,          # an addition
    nkSub,          # a subtraction
    nkIf            # an if statement
  Node = ref object
    case kind: NodeKind  # the ``kind`` field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: Node
    of nkIf:
      condition, thenPart, elsePart: Node

var n = Node(kind: nkFloat, floatVal: 1.0)
# the following statement raises an `FieldError` exception, because
# n.kind's value does not fit:
n.strVal = ""
```

Multi methods

```
type
  Thing = ref object of RootObj
  Unit = ref object of Thing
    x: int

method collide(a, b: Thing) {.inline.} =
  quit "to override!"

method collide(a: Thing, b: Unit) {.inline.} =
  echo "1"

method collide(a: Unit, b: Thing) {.inline.} =
  echo "2"

var a, b: Unit
new a
new b
collide(a, b) # output: 2
```

Meta-programming

- what is it
  - a program that can read, generate, analyze or transform other programs
- why do it
  - can optimise code – by compile-time rewrites
    - think expression templates
  - can enforce better coding patterns
  - can increase code readability and maintainability
    - with great power comes great responsibility
- reflection - when the meta language is the actual language

Meta-programming in Nim

- works on the Abstract Syntax Tree
- respects the type system
- levels of complexity:
  - normal procs and inline iterators
  - generic procs and closure iterators
  - templates
  - macros

Templates

```
1  template withFile(f: untyped, filename: string,
2                    mode: FileMode,
3                    body: untyped): typed =
4    let fn = filename
5    var f: File
6    if open(f, fn, mode):
7      try:
8        body
9      finally:
10       close(f)
11   else:
12     quit("cannot open: " & fn)
13
14 withFile(txt, "ttempl3.txt", fmWrite):
15   txt.writeLine("line 1")
16   txt.writeLine("line 2")
```

AST

```
dumpTree:
  var mt: MyType = MyType(a:123.456, b:"abcdef")

# output:
#   StmtList
#     VarSection
#       IdentDefs
#         Ident "mt"
#         Ident "MyType"
#         ObjConstr
#           Ident "MyType"
#           ExprColonExpr
#             Ident "a"
#             FloatLit 123.456
#           ExprColonExpr
#             Ident "b"
#             StrLit "abcdef"
```

Macros

```nim
import macros

type
  MyType = object
    a: float
    b: string

macro myMacro(arg: untyped): untyped =
  var mt: MyType = MyType(a:123.456, b:"abcdef")
  let mtLit = newLit(mt)

  result = quote do:
    echo `arg`
    echo `mtLit`

myMacro("Hallo")

# The call to myMacro will generate the following code:
echo "Hallo"
echo MyType(a: 123.456'f64, b: "abcdef")
```

More macros

```nim
import macros
dumpAstGen:
  proc hello() =
    echo "hi"
```

```nim
nnkStmtList.newTree(
  nnkProcDef.newTree(
    newIdentNode(!"hello"),
    newEmptyNode(),
    newEmptyNode(),
    nnkFormalParams.newTree(
      newEmptyNode()
    ),
    newEmptyNode(),
    newEmptyNode(),
    nnkStmtList.newTree(
      nnkCommand.newTree(
        newIdentNode(!"echo"),
        newLit("hi")
      )
    )
  )
)
```

More macros - continue from last slide

```
1  import macros
2  macro gen_hello(): typed =
3    result = nnkStmtList.newTree(
4      nnkProcDef.newTree(
5        newIdentNode(!"hello"),
6        newEmptyNode(),
7        newEmptyNode(),
8        nnkFormalParams.newTree(
9          newEmptyNode()
10       ),
11       newEmptyNode(),
12       newEmptyNode(),
13       nnkStmtList.newTree(
14         nnkCommand.newTree(
15           newIdentNode(!"echo"),
16           newLit("hi")
17         )
18       )
19     )
20   )
21 gen_hello()
22 hello() # << same as from last slide!
```

HTML DSL

```
1  import html_dsl
2
3  html page:
4    head:
5      title("Title")
6    body:
7      p("Hello")
8      p("World")
9      dv:
10       p "Example"
11
12 echo render(page())
```

HTML DSL result

```
<!DOCTYPE html>
  <html class='has-navbar-fixed-top' >
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Title</title>
  </head>
  <body class='has-navbar-fixed-top' >
    <p >Hello</p>
    <p >World</p>
    <div>
      <p>Example</p>
    </div>
  </body>
</html>
```

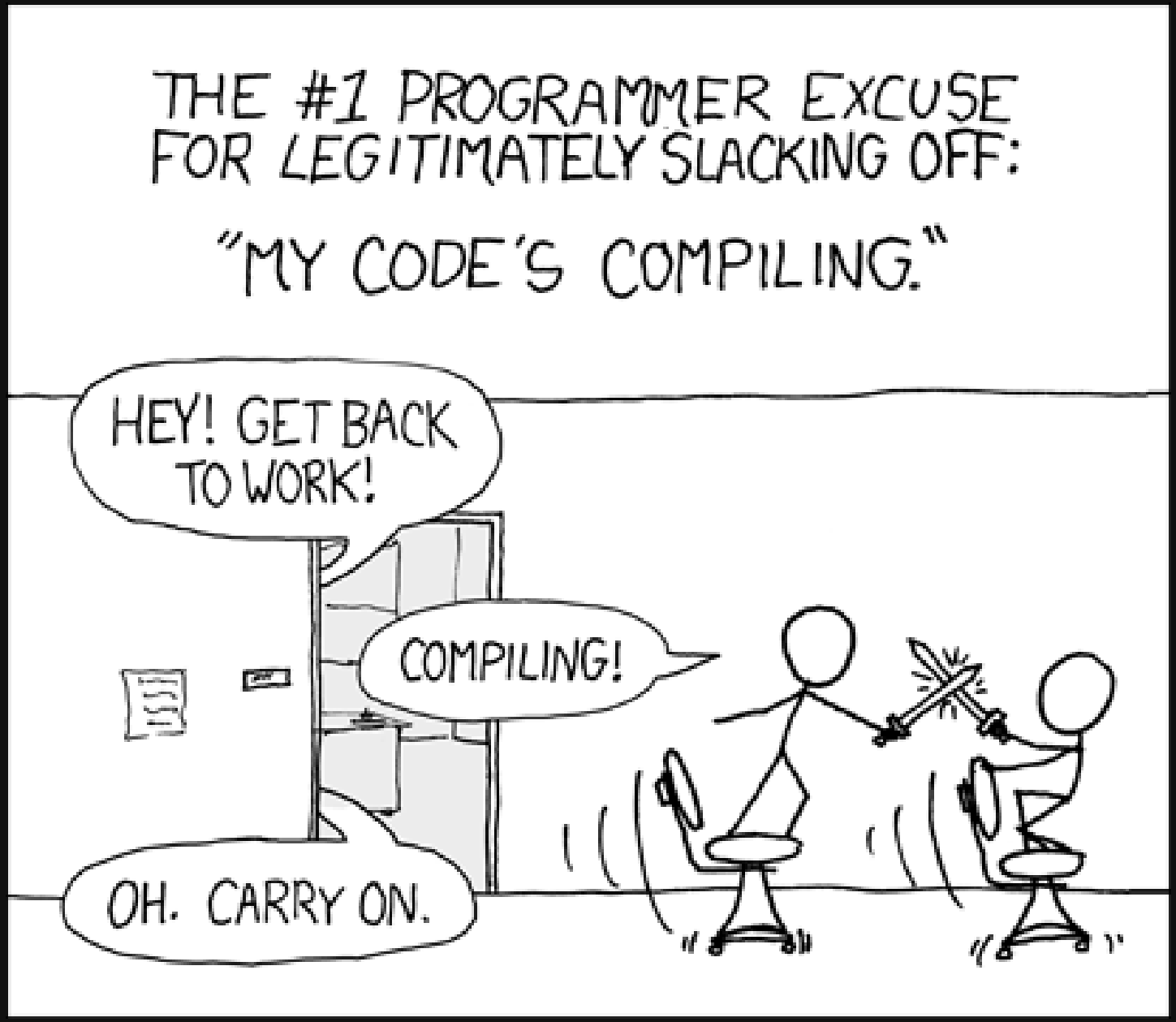
Simply Nim

- statically typed
- high performance (compiles to native binaries - comparable to C/C++)
- very clean & elegant - no, beauty is NOT subjective!
- garbage collected (can do manual memory management too)
- expressive - some of the most powerful metaprogramming
  - compiler has an interpreter inside
- compiles to: C/C++/ObjC/Javascript
  - non-idiomatic - not for reading but optimal for execution
- suited for: systems programming, applications & web
  - all types of software!
- backed by Status since 2018 (#65 cryptocurrency by marketshare)
  - Status - working on one of the first implementations of Ethereum 2.0
  - just like Rust is backed by Mozilla (although with a lot less...)
- has a rich stdlib, package manager, docs, some IDE support

Feature rundown

- uniform call syntax (extension methods) - obj.method() OR method(obj)
  - that's why there are no real "methods" defined in types
- function call parens are optional - echo("hello") OR echo "hello"
- case-insensitive - also underscore-insensitive but that's another topic :|
- generics
- templates (meta-programming^2)
- macros (meta-programming^3) - evaluated in the compiler by the NimVM
- concepts
- discriminated unions
- strong typedefs (distinct type) - can has $ currency?
- coroutines & closures
- switch & pattern matching
- dynamic dispatch & multi-methods
- converters - explicit (for implicit conversions)
- effect system (transitive)
- extensible pragmas, "defer", exceptions, "discard", named args... good defaults!

My "favourite" aspect of C++



A bit on C++

- C++20 is shaping up to be a huge release
  - lots of cool stuff, but complexity is through the roof
  - Expert-"tolerable" - prestige when you come up with yet more complicated TMP
- simple example using ranges from C++20 - blog post
  - 3 seconds of compile time for ~20 lines of code, forget about "Debug" builds
- Remember the Vasa! - Bjarne Stroustrup
- There should come a time for a clean slate
  - C++ is a great and valuable ongoing research
  - The 2 biggest reasons C++ is so widely used today:
    - legacy and maturity - too much software written already
    - inertia - attachment and lack of interest to learn new languages
  - C++ is a HUGE time/money cost on the scale of hundreds of millions
    - developer productivity, bug & safety
    - business should back a better language & push for development + learning

Some quotes & thoughts

- Fifty years of programming language research, and we end up with C++?
  - Richard A. O'Keefe
- There are only two kinds of programming languages: those people always bitch about and those nobody uses.
  - Bjarne Stroustrup
- Nim is the next iteration of practical language design
  - by humble !!! >> me << !!!
- Nim: speed of C, elegance of Python, flexibility of Perl
  - Peter Munch-Ellingsen
- Nim is to C++ as CoffeeScript is to JavaScript
  - cjhanks, hackernews Apr 18, 2017

Comparison with others

- D, Rust, Jai, Zig
  - out of scope for this talk
- Go
  - not really a *pinnacle* of abstraction and innovation :|
- C++
  - <optional> - 5k+ LOC for a T and a bool... safe_int - same horror story
  - The next big thing: "Design by introspection" - Andrei Alexandrescu
- Nim is one of the most logical paths forward
  - on-par performance with C/C++ (compiles to them)
  - some of the most easy interop with C/C++ ..... (compiles to them)
  - uses any C/C++ compiler ..... (compiles to them)
  - already quite far in terms of implementation
  - meta-programming on steroids

Nim compilation model

```nim
# main.nim

import a

echo a()
```

```nim
# a.nim

import b

proc a*(): string =
  return from_b
```

```nim
# b.nim

let local = "B!"

let from_b* = local
              ^
# means "exported"
```

- nim c -d:release main.nim
  - always compile only the main file, follow the imports
  - whole program analysis
  - a .c file for each .nim file in a "nimcache" (temp) folder (also .obj files)
  - only referenced (imported) modules are compiled in the end
- entire project is always "compiled" by Nim (currently no "minimal" rebuild)
  - ~4-5 sec for the entire source of Nim - 135 files (without the C compiler)
  - the C/C++ compiler rebuilds only changed files (takes a bit more time)
  - will change when per-module caching is introduced - even faster!

Nim to C/C++: nimbase.h

included by all .c/.cpp files

```
1  // nimbase.h
2
3  #define N_NIMCALL(rettype, name) rettype __fastcall name
4  #define N_CDECL(rettype, name) rettype __cdecl name
5  //...
6  #define N_NIMCALL_PTR(rettype, name) rettype (__fastcall *name)
7  //...
8  #define N_LIB_PRIVATE __attribute__((visibility("hidden")))
9  //...
10 #define N_LIB_EXPORT  extern __declspec(dllexport)
11 //...
12 #define STRING_LITERAL(name, str, length) \
13    static const struct {                  \
14      TGenericSeq Sup;                     \
15      char data[(length) + 1];             \
16   } name = {{length, (int) ((unsigned)length | NIM_STRLIT_FLAG)}, str}
```

handles different platforms - convenience macros

Nim procs to C/C++

```nim
proc foo() =
  echo "hello"

foo()
```

```c
#include <nimbase.h>

// forward declarations / type definitions / constants section
struct TGenericSeq { int len; int reserved; };
struct NimStringDesc : public TGenericSeq { ... };
typedef NimStringDesc* tyArray_nHXaesL0DJZHyVS07ARPRA[1];

STRING_LITERAL(TM_r9bkcJ6PRJ5n7ORNxxJ5ryg_3, "hello", 5); // << string litera
NIM_CONST tyArray_nHXaesL0DJZHyVS07ARPRA TM_r9bkcJ6PRJ5n7ORNxxJ5ryg_2 =
    {((NimStringDesc*) &TM_r9bkcJ6PRJ5n7ORNxxJ5ryg_3)};

N_LIB_PRIVATE N_NIMCALL(void, foo_iineYNh8S9cE6Ry7dr2Tz2A)(void); // << fwd d

// definition section
N_LIB_PRIVATE N_NIMCALL(void, foo_iineYNh8S9cE6Ry7dr2Tz2A)(void) { // << def
    echoBinSafe(TM_r9bkcJ6PRJ5n7ORNxxJ5ryg_2, 1); // the echo call
}

// code execution section
foo_iineYNh8S9cE6Ry7dr2Tz2A(); // << call
```

Nim types to C/C++

```
type
  MyData = object
    answer: int
    ready: bool
proc newData(): MyData = return MyData(answer: 42, ready: true)
echo newData().answer
```

```
// forward declarations / type definitions / constants section
struct tyObject_MyData {
    int answer;
    bool ready;
};
// definition section
N_LIB_PRIVATE N_NIMCALL(tyObject_MyData, newData)(void) {
    tyObject_MyData result; // always an implicit "result"
    nimZeroMem((void*)(&result), sizeof(tyObject_MyData));
    result.answer = ((int) 42);
    result.ready = true;
    return result;
}

// code execution section
tyObject_MyData T2_;
T2_ = newData(); // << call
//...
```

Nim closures to C/C++ (resumable funcs)

```
1 iterator closure_iter*(): int {.closure.} = # a resumable function
2   var x = 1
3   while x < 10:
4     yield x
5     inc x
6 for i in closure_iter(): echo i
```

```
 1 struct state_type : public RootObj {
 2     int colonstate_; // state progress - there are some GOTOs using this
 3     int x1; // the state
 4 };
 5
 6 struct closure_type {
 7     N_NIMCALL_PTR(int, c_ptr) (void* e_ptr); // function ptr
 8     void* e_ptr; // environment ptr
 9 };
10
11 N_LIB_PRIVATE N_CLOSURE(int, func)(void* e_ptr) { // def omitted for simplici
12
13 state_type st; // the state
14 closure_type local; // the closure
15 local.c_ptr = func; // assign the func
16 local.e_ptr = &st; // assign environment
17 //...
18 i = local.c_ptr(local.e_ptr); // the call in the loop
```

Nim compilation to C/C++: a BIG win

- smaller scope for the compiler
- all the cutting-edge optimization for C/C++ for free
- out-of-the-box support for tons of platforms
- easiest C/C++ interop possible
- exceptions - reusing those of C++ when using that backend
- nim to C/C++ code mapping with #line directives for debuggers
- no generated headers for the exported parts of modules
- each .c/.cpp file contains everything (and only what) it needs
  - forward declarations for external functions
  - type definitions
- each .c/.cpp file includes nimbase.h and a few C stdlib headers
- high level macros & templates => simple structs and functions

Interfacing with C/C++

Foreign Function Interface

```
proc printf(formatstr: cstring)
    {.header: "<stdio.h>", importc: "printf", varargs.}
```

other pragmas - for use in Nim:

```
{.emit: """
using namespace core;
""".}

{.compile: "logic.c".}
```

We can also call Nim code from C/C++:

```
# fib.nim
proc fib(a: cint): cint {.exportc.} # do not mangle
```

```
nim c --noMain --noLinking --header:fib.h fib.nim
```

```
// user.c
#include <fib.h>
```

Interfacing with C/C++

C++ template constructs

```
type
  StdMap {.importcpp: "std::map", header: "<map>".} [K, V] = object
proc `[]=`[K, V](this: var StdMap[K, V]; key: K; val: V) {.
  importcpp: "#[#] = #", header: "<map>".}

var x: StdMap[cint, cdouble]
x[6] = 91.4
```

Generated C++

```
std::map<int, double> x;
x[6] = 91.4;
```

c2nim tool - generate C/C++ bindings for Nim

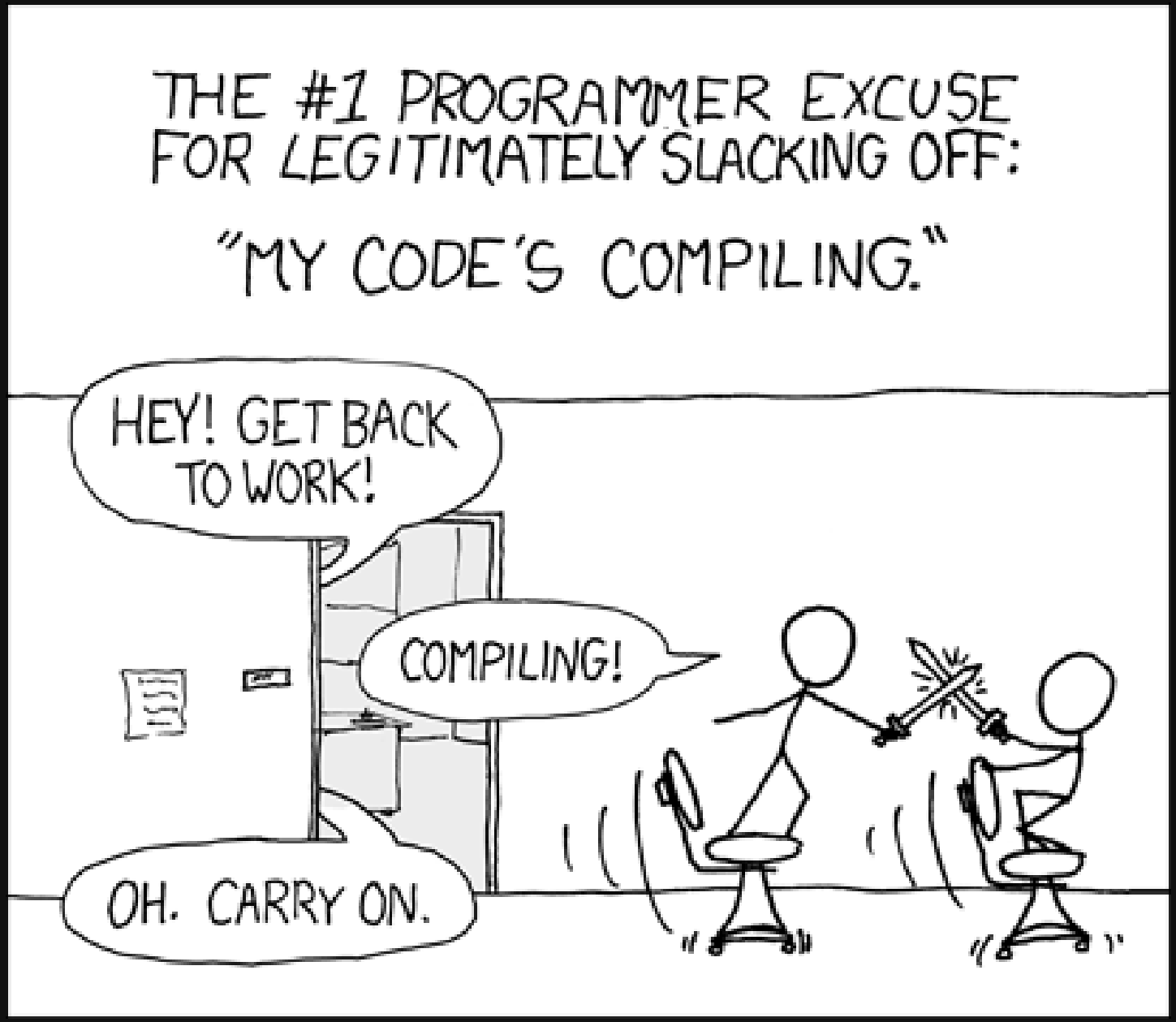
Runtime compilation - WHY

- much faster iteration times
  - no need to restart the program - can preserve state
- less need for a scripting language
  - no need for a virtual machine
  - no binding layer
  - code in one language
- can hack something quickly
  - introspection, queries
    - debuggers aren't infinitely powerful
  - fine-tuning values
- interactive (REPL-like): very useful for exploration and teaching

Runtime compilation for C/C++: HOW

- replacing entire functions: using shared libraries OR hot-patching:
  - possible for decades - but not widely used
  - usually quite intrusive (interfaces, constraints, complicated setup)
  - in game engines: Unreal, others...
  - hot-patching (with very little setup): Live++, Recode
  - Visual Studio "Edit & Continue" - 0 setup, but limited
  - https://github.com/crosire/blink
  - https://github.com/ddovod/jet-live
  - http://bit.ly/runtime-compilation-alternatives << "one link to rule them all"
- interactive: REPL-like
  - cling - by researchers at CERN - built on top of LLVM
    - inspector, Jupiter
    - hard to integrate in a platform/compiler agnostic way
  - RCRL - basically a hack - the inspiration for the Nim implementation

Replace "compiling" with "restarting"



Hot code-reloading (HCR) in Nim

- inspired by a hacky REPL for C++ (called RCRL - by me)
- https://github.com/nim-lang/Nim/issues/8927
  - mentored by Zahary
- compile with --hotCodeReloading:on
- need also 2 .dlls (the HCR runtime + the GC of Nim)

```nim
# main.nim

import hotcodereloading # for reload
import other

while true:
  echo readLine(stdin) # pause
  performCodeReload()  # reload
  echo getInt()        # call
```

built as an .exe/.dll depending on the project type

```nim
# other.nim

import hotcodereloading # for after handler

var glob = 42

proc getInt*(): int = return glob + 1 # exported

afterCodeReload:
  glob = 666
```

built as a reloadable .dll ends up in the "nimcache"

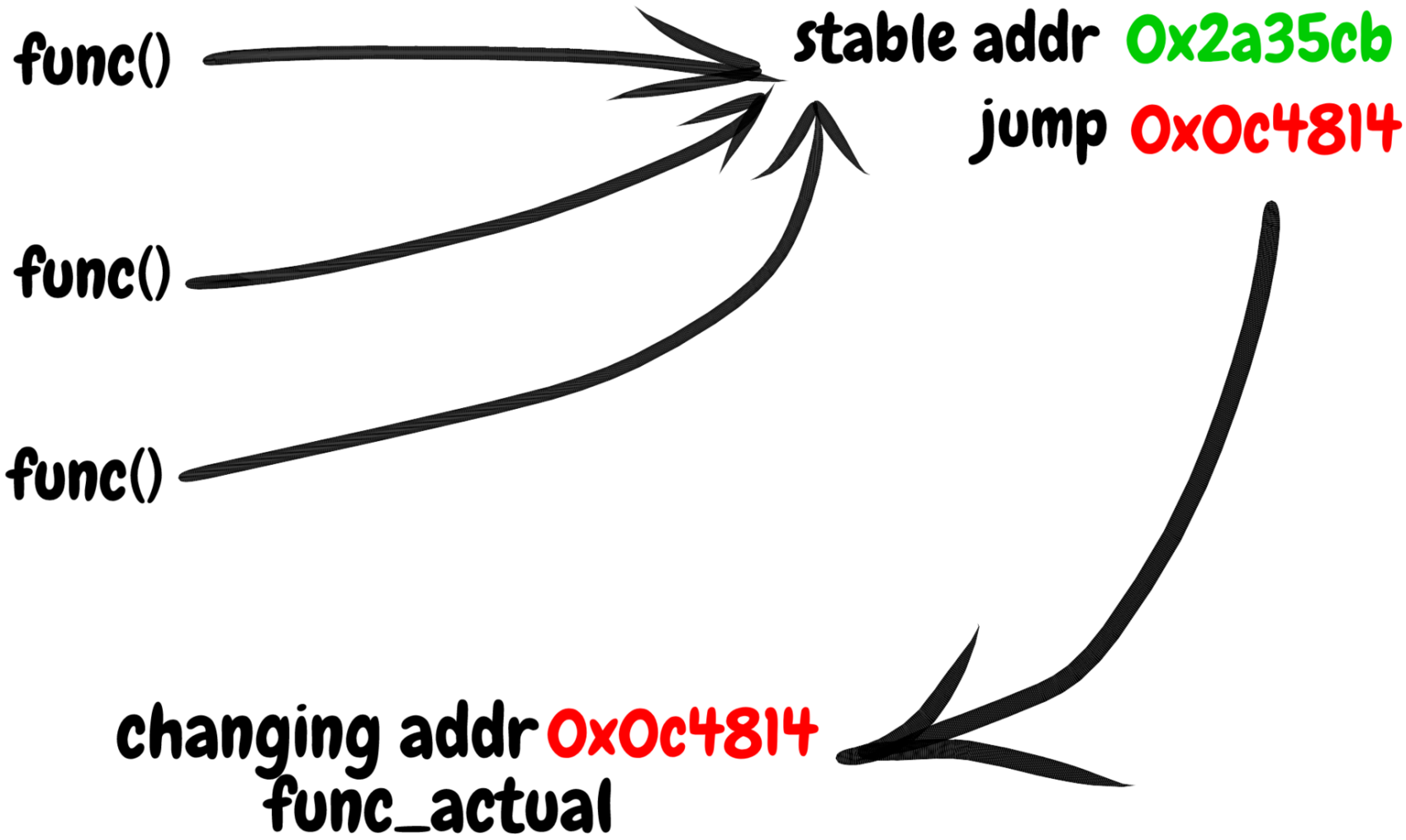
Effects of HCR

- all interaction between .nim modules => through pointers
- functions - changes:
  - forward declarations become function pointers
  - definitions get "_actual" as a suffix
  - pointers are assigned the "_actual" on startup
  - calls stay the same (pointer has the same name)
- globals - changes:
  - turned into pointers
  - allocated on the heap and initialized on startup
    - state is preserved when reloading
  - dereferenced wherever used

Effects of HCR

```
1  // fwd decl/globals section
2  static N_NIMCALL_PTR(int, getInt_omy6T2FkprLEReOy2ITmIQ)(void);
3  static int* glob_v1zK9aUOu9aNNcsxruuK8NdA;
4
5  // definitions
6  N_LIB_PRIVATE N_NIMCALL(int, getInt_omy6T2FkprLEReOy2ITmIQ_actual)(void) {
7          int result;                                          // ^^ the suffix
8          result = (*glob_v1zK9aUOu9aNNcsxruuK8NdA);
9          return result;
10 }
11
12 // usage
13 (*glob_v1zK9aUOu9aNNcsxruuK8NdA) = getInt_omy6T2FkprLEReOy2ITmIQ();
14
15 // init on startup (naive)
16 glob_v1zK9aUOu9aNNcsxruuK8NdA = new int(42);
17 getInt_omy6T2FkprLEReOy2ITmIQ = getInt_omy6T2FkprLEReOy2ITmIQ_actual
```

Trampolines



Initialization

```
1 // naive
2 glob_v1zK9aUOu9aNNcsxruuK8NdA = new int(42);
3 getInt_omy6T2FkprLEReOy2ITmIQ = getInt_omy6T2FkprLEReOy2ITmIQ_actual
```

```
 1 // reality
 2 getInt_omy6T2FkprLEReOy2ITmIQ = (tyProc_vVu2P82aVLv9c8X0xbI1NJw) hcrRegisterProc(
 3     "D:\\play\\nimcache/play.cpp.dll",            // "domain" (AKA module)
 4     "getInt_omy6T2FkprLEReOy2ITmIQ",              // "key"
 5     (void*)getInt_omy6T2FkprLEReOy2ITmIQ_actual); // the real function
 6
 7 if(hcrRegisterGlobal("D:\\play\\nimcache/play.cpp.dll",        // "domain" (AKA module)
 8                       "glob_v1zK9aUOu9aNNcsxruuK8NdA",        // "key"
 9                       sizeof((*glob_v1zK9aUOu9aNNcsxruuK8NdA)), // size for allocation
10                       NULL, // for the GC - simple integer is simple, so NULL
11                       (void**)&glob_v1zK9aUOu9aNNcsxruuK8NdA))  // address to pointer
12 {
13     // hcrRegisterGlobal returns "true" only if not already inited
14     (*glob_v1zK9aUOu9aNNcsxruuK8NdA) = ((int) 42); // init with value (or side effects)
15 }
```

Initialization

- the HCR.dll runtime holds pointers to all globals/functions
- hcrRegisterProc
  - allocates executable memory (a few bytes)
  - writes a jump instruction (trampoline) to the "_actual"
  - returns an address to the trampoline
  - this way "_actual" can be changed on reloading
    - changed by calling it again with a different address
    - all pointers to the trampoline stay the same
- all symbols are registered per "domain" (.dll)
  - no name clashes (even though they are mangled...)
  - better management - can remove all symbols for module X

Initialization

```nim
# main.nim

import a, b

echo from_a()
echo from_b()
```

```nim
# a.nim

import b

proc from_a*(): string =
  result = "A!"
  result.add from_b()
```

```nim
# b.nim

proc from_b*(): string =
  return "B!"
```

1. main.exe loads the hcr.dll (and the Nim GC in rtl.dll)
2. main.exe calls init() from hcr.dll and passes a list of imports (a, b)
3. hcr.dll loads a.dll and gets a list of imports (b)
4. hcr.dll loads b.dll and fully initializes it (it has no imports)
    1. registers from_b() and does nothing else
5. hcr.dll fully initializes a.dll
    1. registers from_a() and gets the address for from_b()
6. hcr.dll skips b.dll (part of the imports of main.exe) since it is already initialized
7. main.exe is initialized
    1. gets the addresses for from_a() and from_b()
    2. executes the top-level code (the 2 echo statements)

Initialization

- a DFS traversal with POST visit
- when module A imports a symbol from B
  - symbol is first registered in B
  - symbol is "gotten" in A after B is inited
- basically a custom dynamic linker :|
- imports are discovered on-the-go
- HCR.dll constructs a tree of imports and maintains it
- many details omitted
  - initialization is broken into multiple passes
    - registration of type infos (for the GC) is a pre-pass
- each .dll exports just a few functions which the HCR.dll uses
  - getImports(), and the ones for the passes

Reloading

when we call performCodeReload():

- HCR.dll will check hasAnyModuleChanged()
  - basically scanning if any .dll has been modified (timestamp)
- changes shouldn't affect .dll files which are part of the current active callstack when reload() is called! or crash :|
  - ==> main module can never be reloaded
- execute the "beforeCodeReload" handlers if about to reload
- in a DFS traversal, for each modified module:
  - same as the init - get its imports, load them (if changed or new), init everything in proper order
    - supports discovery of new imports!
    - also removes no longer referenced modules and their symbols
- execute the "afterCodeReload" handlers

Reloading - handlers

```nim
# main.nim
import a, b, hotcodereloading

beforeCodeReload:
  echo "before main"
afterCodeReload:
  echo "after main"
```

```nim
# a.nim
import b, hotcodereloading

beforeCodeReload:
  echo "before a"
afterCodeReload:
  echo "after a"
```

```nim
# b.nim
import hotcodereloading

beforeCodeReload:
  echo "before b"
afterCodeReload:
  echo "after b"
```

- DFS traversal with POST visit
- handlers can be added/removed
- can be used to update globals
- fine-grained control:
  - hasModuleChanged(<module>)

only A changes => all handlers are executed on reload:

```
before b
before a
before main
after b
after a
after main
```

Reloading - global scope

- top-level code (global scope) is executed only on initial load
  - for new top-level code use before/after handlers
- changing the initializer of a global doesn't do anything
  - use a before/after handler
  - or remove the global entirely, reload, and re-add it
    - brand new symbol!
- new globals can be added - and will be initialized properly

The initial HCR example revisited

```nim
# main.nim

import hotcodereloading # for reload
import other

while true:
  echo readLine(stdin) # pause
  performCodeReload() # reload
  echo getInt() # call
```

```nim
# other.nim

import hotcodereloading # for after handler

var glob = 42

proc getInt*(): int = return glob # exported

afterCodeReload:
  glob = 666
```

Makes more sense now, doesn't it?

Encountered problems

- processes lock loaded .dll files in the filesystem on Windows
  - when reloading we copy x.dll to x_copy.dll and load the copy
- changing module X can affect module Y
  - such changes shouldn't reach the main module
  - mangling of symbols being affected by attributes (purity)
  - mangling affected by where "inline" functions get used first
  - mangling affected by which module instantiates a generic
- C vs C++
  - missing forward declarations - fine in C!
  - multiple identical forward declarations
    - multiple definitions of global function pointers - fine in C!

Visual Studio debug symbols - PDB drama

- .dll/.exe have hardcoded paths to the .pdb (copying the .dll doesn't matter)
- the VS Debugger keeps the .pdb files locked for .dlls even after unloaded

solutions:

- someone managed to close the file handles to no longer needed .pdb files (.dll has been unloaded) to the external VS debugger process (live++)
- embed the debug info in the actual binaries just like on unix
  - /Z7 embeds it in .obj files but not for the final .dll/.exe when linking them
- different names for the .pdb using /PDB:<filename> (with the date/time (including milliseconds) as a suffix)
  - the "hardcoded" paths to .pdb files are always different
  - try to delete all <dll_name>_*.pdb files for a given .dll when linking
    - failure to delete them means the VS debugger still holds them locked
    - links: l1, l2, l3, l4

HCR performance

- snappy compression algorithm - x2-x4 times slower
  - for reference: zlib (c code) to javascript (asm.js) ==> x2 slow down
- calls within a translation unit are direct (the "_actual" version gets called)
- calls between modules => indirection: pointer to function
  - + additional jump from trampoline to actual function
- link time optimization (AKA whole program optimization) cannot help
  - devirtualization techniques are not applicable either
- compactness in memory VS a single binary => instruction cache misses
- /hotpatch for MSVC and Live++ (which are faster):
  - not going through function pointers
  - by default there are no jumps in the function preamble (padding)
- slowdown depends a lot on the type/scale of software - x2 to x5...

HCR performance

possible optimizations:

- write more "inline" procs
  - their body is emitted wherever used => skip indirections
- pragmas for excluding files (extension of the first point in this list)
  - register the module procs but no indirections between them
- relocate all code from loaded binaries close in memory?
- PLOT TWIST!
  - debug builds are currently affected a lot less (<x2 slowdown)
    - HCR is mainly for development => probably debug builds

HCR TODO

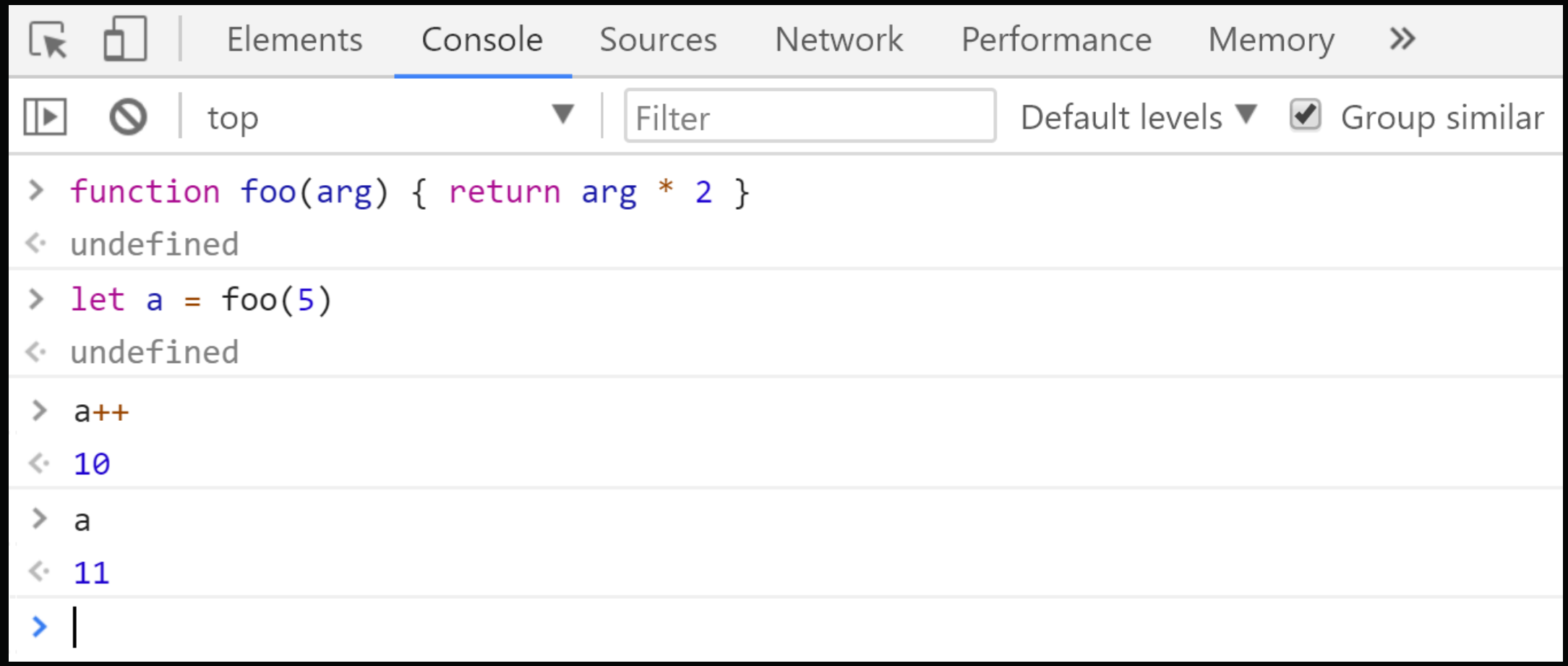
- Nim stdlib has trouble compiling with the GC as a separate SO
  - "-d:useNimRtl" needs to be enabled for all compiler tests
  - currently no real-world project can be built with HCR
- detecting type changes
  - error when detected
  - OR ability for users to handle it (migrate data)
- check if "reload" would affect functions from the current call stack
- expose state for outside manipulation with interactive speeds
  - imagine a slider in the IDE for a variable or a color picker widget
- performance & bug fixes

HCR Implementation choice

- pros
  - any modern (desktop) OS supports dynamic libraries
  - works with any C/C++ compiler
  - near-native speeds
  - final binaries are debuggable
  - a REPL is easily built on top of this
  - (arguably) less complex than using LLVM / JIT / whatever
  - changes are isolated (only the C backend which is a few files)
  - program can be changed in (almost) any way
  - novel approach - someone had to try it
- cons
  - not as optimal as the /hotpatch for MSVC or Live++
  - (arguably) more complex than using LLVM / JIT / whatever
  - not sure how NLVM (Nim on top of LLVM) will support HCR

REPL - Read Eval Print Loop

- interpreted languages have it (JavaScript, Python, etc.)
- consoles/shells - cmd.exe, bash
- can iteratively append/execute code (definitions, side effects, etc.)
- education, scientific community, rapid prototyping of any kind



REPL/Nim quote

Nim is the language I have always thought was a brilliant idea that I never get to use. It's a shame.

Nim is to C/C++ as CoffeeScript is to JavaScript. A highly extensible template language atop a portable language with libraries for practically everything.

So why haven't I hopped on the bandwagon? Outside of C++, C, and Fortran - the only way I have ever learned a new language is through using a REPL. How much of Python's and MATLAB's (and maybe even Julia's) success is due to having a brilliant REPL?

I am not complaining, and I do not have any free time to fix it. But man... if Nim just had a killer REPL that allowed me to slowly learn the language properly while not being blocked from my daily work... it would be just killer!

cjhanks on Apr 18, 2017

https://news.ycombinator.com/item?id=14143521

REPL on top of HCR

Talk abstract was a lie! didn't get to implementing it in time...

2 files:

- main module
  - has the main loop
  - handles code submissions
- imported file
  - gets modified based on submissions
  - rebuilt + reloaded

should be well below half a second

REPL on top of HCR

you submit this:

```
import tables

var a = {1: "one", 2: "two"}.toTable

echo a
```

and it gets translated to this:

```
import hotcodereloading # for the before/after handlers

import tables

var a = {1: "one", 2: "two"}.toTable

afterCodeReload:
  echo a
```

REPL on top of HCR

later you append:

```
let b = a

echo b
```

and it gets translated to this:

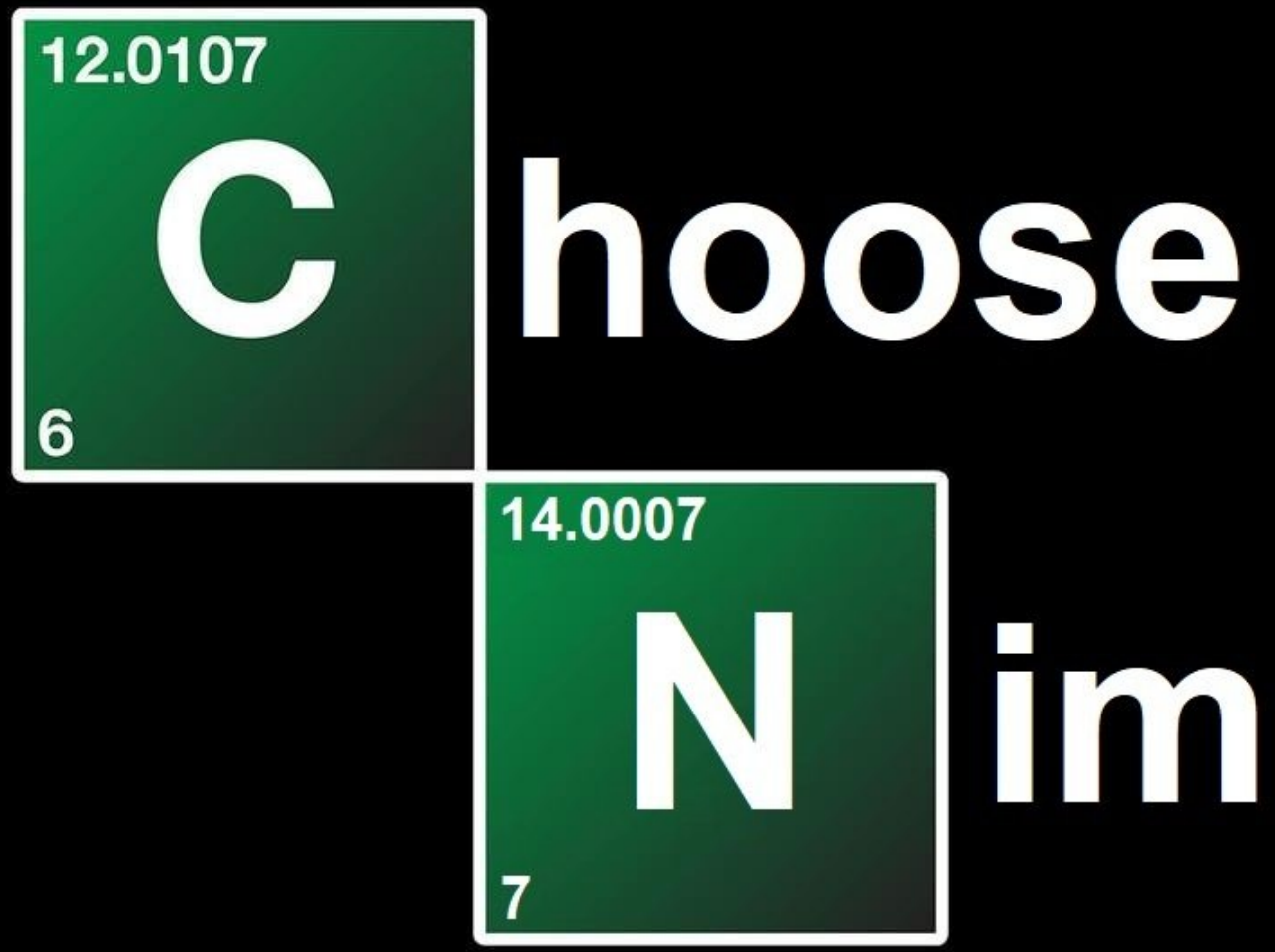
```
import hotcodereloading # for the before/after handlers

import tables

var a = {1: "one", 2: "two"}.toTable

let b = a # the new code

# only the new side effects are still present
afterCodeReload:
  echo b
```

Jupyter kernel

- yesterday on ACCU: Interactive C++ : Meet Jupyter / Cling - The data scientist’s geeky younger sibling - by Neil Horlock
- A Jupyter Notebook is an interactive document - a collaborative platform for prototyping, experimentation and analysis
- Mix and share: code, text, data, computation and visualization
- "Notebooks are the most popular tool for working with data at Netflix."
- Nim REPL => Nim Jupyter kernel

The road ahead for Nim

- version 1.0 - promise of stability
- compiler cache for unchanged modules
  - because compilation starts always from the main module
  - of great benefit for HCR/REPL
- more features
- better tooling
- better docs
- taking over the world
- get involved - still in early stages - you can have an impact


12.0107
C
6
hoose
14.0007
N
7
im

Q&A

- https://nim-lang.org/
- https://github.com/nim-lang/Nim
- FOSDEM 2019: Metaprogramming with Nim

- Slides: https://slides.com/onqtam/nim_hot_code_reloading
- Blog: http://onqtam.com
- GitHub: https://github.com/onqtam
- Twitter: @KirilovVik
- E-Mail: vik.kirilov@gmail.com