Challenges using LLVM for OCaml

A rant about garbage collection in LLVM

Stephen Dolan

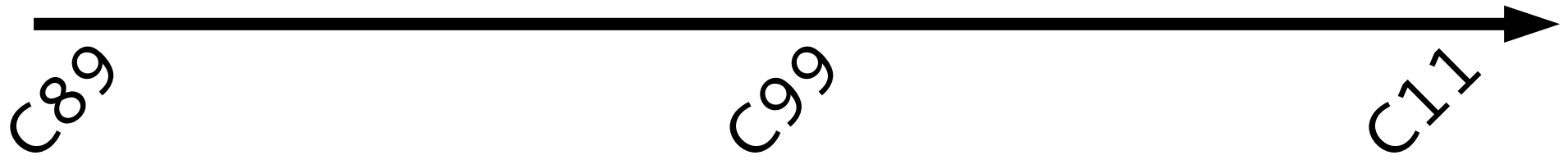
Cambridge LLVM Day

Monday 18th November 2013

LLVM

is a general purpose compiler framework

that can handle the entire spectrum of programming languages



Garbage collection

is a modern* technique for memory management

used in a couple of research languages
(not ones used for Real Programming, of course)

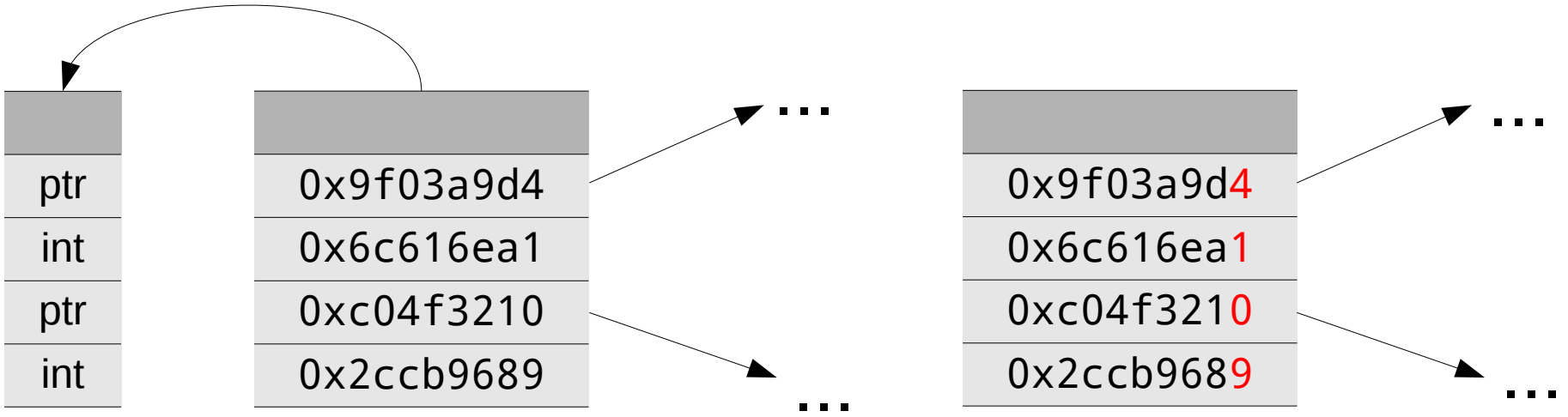
*McCarthy, 1959

Garbage collection

Automatically free memory
when it's no longer referenced

To do this, we need to find the pointers

Finding pointers in the heap



Tables
(Haskell)

Tags
(OCaml)

How do we find the pointers on the stack,
when the compiler doesn't tell us where they are?

Conservative GC

How do we find the ~~pointers~~ stuff on the stack,
when the compiler doesn't tell us where they are?

Conservative GC

- Scan everything, looking for “pointers”
  - can leak when non-pointers look like pointers
- GC can never move objects
  - kills most of the good GC algorithms

Separate stack

How do we find the pointers ~~on the stack~~ in some array,
when the compiler doesn't tell us where they are?

Separate stack

```
f :: String -> [String] -> String
f s l = if elem s l
          then s ++ " was found!"
          else s ++ " wasn't found."
```

```
movq    %rcx, -8(%rbp)
movq    $sn4_info, -16(%rbp)
movq    %rdx, %rbp
movl    $M_fzuzddEq_closure, %r14d
movq    %rcx, %rsi
movq    %rax, %rdi
jmp     base_GHCziList_elem_info
```

push s onto GHC's stack
(which is an array of int64)

tail-call “elem”

Separate stack

- Separate function for every block
  - defeats LLVM intra-procedual optimisations
- Locals often end up on GHC's stack
  - defeats LLVM local variable optimisations
- All calls done via `jmp`
  - defeats hardware return prediction

Shadow stack

- Keep variables on the normal stack, but also put a copy elsewhere for the GC to find.
  - LLVM has support for this
  - Doesn't defeat optimisations quite as much
  - Lots of overhead

Stack maps

How do we find the pointers on the stack,
when the compiler ~~doesn't~~ does tell us where they are?

Stack maps

```
let f (s : string) (l : string list) =
  if List.mem s l
  then "a" ^ " was found!"
  else "a" ^ " wasn't found."
```

```
    call    camlList__mem_1156@PLT
.L102:

    ...

camlM__frametable:
    .quad   1
    .quad   .L102
    .word   16
    .word   0
```

do a real call

stack map as static data

Stack maps

- No overhead while not GC-ing
  - compiled as static data about the code
- Compiler must tell runtime where values are
  - Compiler must keep track of values through optimisations

Stack maps in LLVM

`@llvm.gcroot`

- Per-function, not per return address
  - extra overhead clearing slots
  - buggy interactions with inlining (#16778)
- Can't express that a register is a root
  - has to spill everything anywhere GC could happen
- No builtin support for actually generating a map
  - “implement your own plugin”

</rant>

- LLVM *nearly* supports efficient GC
  - but @llvm.gcroot is a poor interface
  - and it's buggy
  - and you have to write a nontrivial LLVM extension to use it

- Questions? Counter-rants?