Remote GHCi

Simon Marlow

Haskell Implementors Workshop 2016

Déjà vu

- HIW 2012: “Why can’t I get a stack trace?”
- HIW 2016: now you have 3 ways to get a stack trace…
  (that must be better, right?)

- This talk:
  - We now have always-on stack traces in GHCi!
  - … and that solved some other problems at the same time

Background

- Haxl at Facebook
  - DSL for writing abuse-detection code
  - Runs at scale: 10^6 requests/second
  - Engineers in anti-abuse teams need to develop and test their Haxl code
  - So we built a customized GHCi, called haxlsh

haxlsh is our IDE

- In haxlsh, developers can
  - Load their code and test it against live data
  - Reproduce and debug problems that occurred in production

- But… debugging can be hard

Exceptions

```
haxlsh> myFunction x y z
*** Exception: NotFound (“field \”wibble\” was not found”)
```

- To debug this, the programmer would have to look at the definition of `myFunction` and run the pieces separately
- And keep digging until they find the culprit

“Why can’t I get a stack trace?”

- “Because you need to use profiling”
- “Or you need to use `HasCallStack` everywhere”
- “Or you need DWARF”
  - (which isn’t working yet)

“Why can’t I get a stack trace?”

- **“Because you need to use profiling”**
- “Or you need to use `HasCallStack` everywhere”
- “Or you need DWARF”
  - (which isn’t working yet)

Demo

Towards GHCi + profiling

- The profiler keeps a call stack
- How do we use the profiler's stack in GHCi?

1. Build GHCi against the profiled RTS
2. Load profiled packages
3. profit


PHABRICATOR
Differential > D1407
Make GHCi & TH work when the compiler is built with -prof
Closed
Public
Authored by simonmar on Oct 30 2015, 6:33 PM.

Ok, but we still need stacks

- In compiled code we use `-fprof-auto` to add annotations
- The interpreter already adds annotations everywhere for breakpoints
  - Common annotation pass used by profiling, hpc, breakpoints
- We can re-use those annotations for stacks
- … just need to hook it up

Differential > D1595

Maintain cost-centre stacks in the interpreter

Closed Public

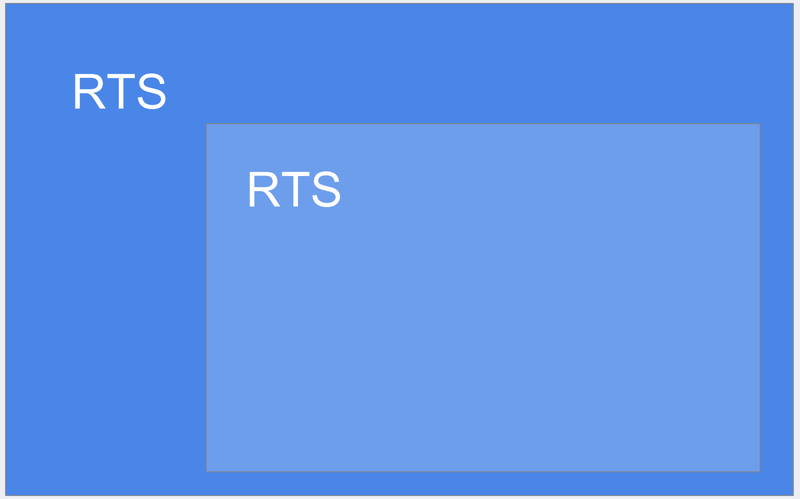
Authored by **simonmar** on Dec 10 2015, 7:34 AM.

It worked...

- But profiling is optional for a reason: It's *really slow*.
- GHCi was 2x-3x slower at loading code
  - (matters when you have thousands of modules)
- We need:
  - Full speed compiler
  - Interpreted code running with profiling

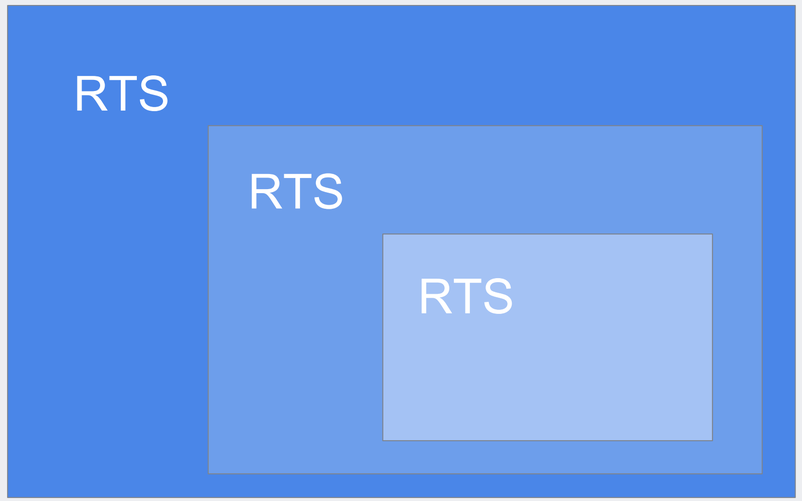
Two runtimes in the same process?

- The RTS could load another RTS using its own linker



Two runtimes in the same process?

- The RTS could load another RTS using its own linker



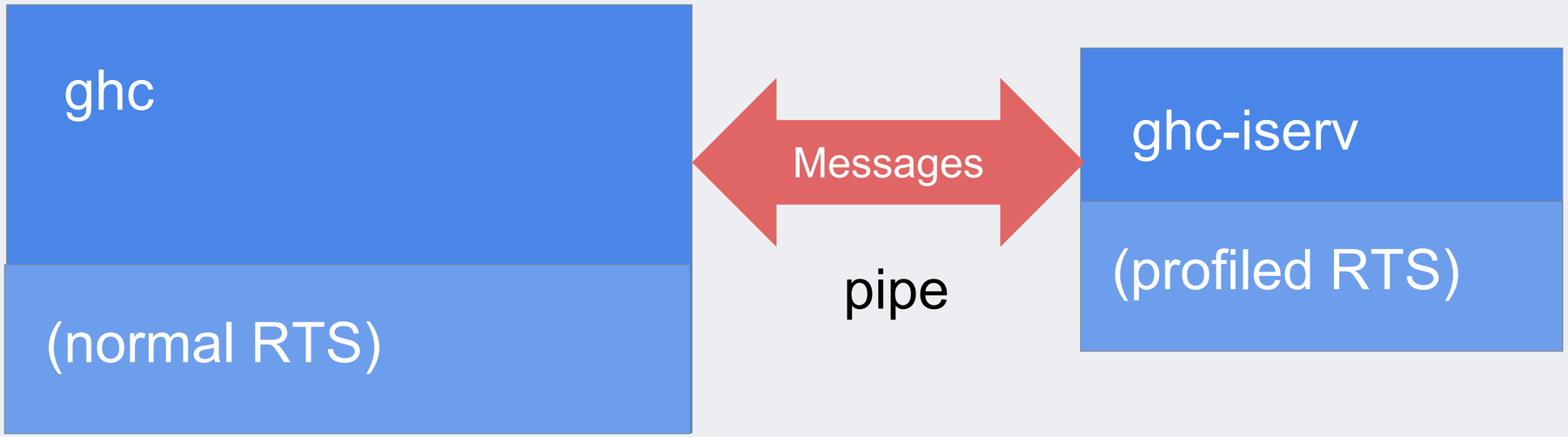
But...

- We still have to communicate between the different instances somehow
- Can’t share data between the heaps
- Ultimately it’s simpler to have separate processes
  - Easier to debug and manage

Why not side-by-side?

- Could a single RTS have multiple “instances”?
  - Each with its own heap, scheduler, etc.
- In theory yes, but a lot of work
  - Couldn’t share compiled code (because CAFs)
  - Profiling is a compile-time choice in the RTS right now

Remote GHCi



How does Remote GHCi work?

1. Define an API between GHC and the interpreter
2. Implement the API via RPCs over the pipe

The API contains

- The runtime linker (load an object, resolve a symbol)
- Create byte-code objects
- Evaluate things
- Template Haskell...

Typed binary messages

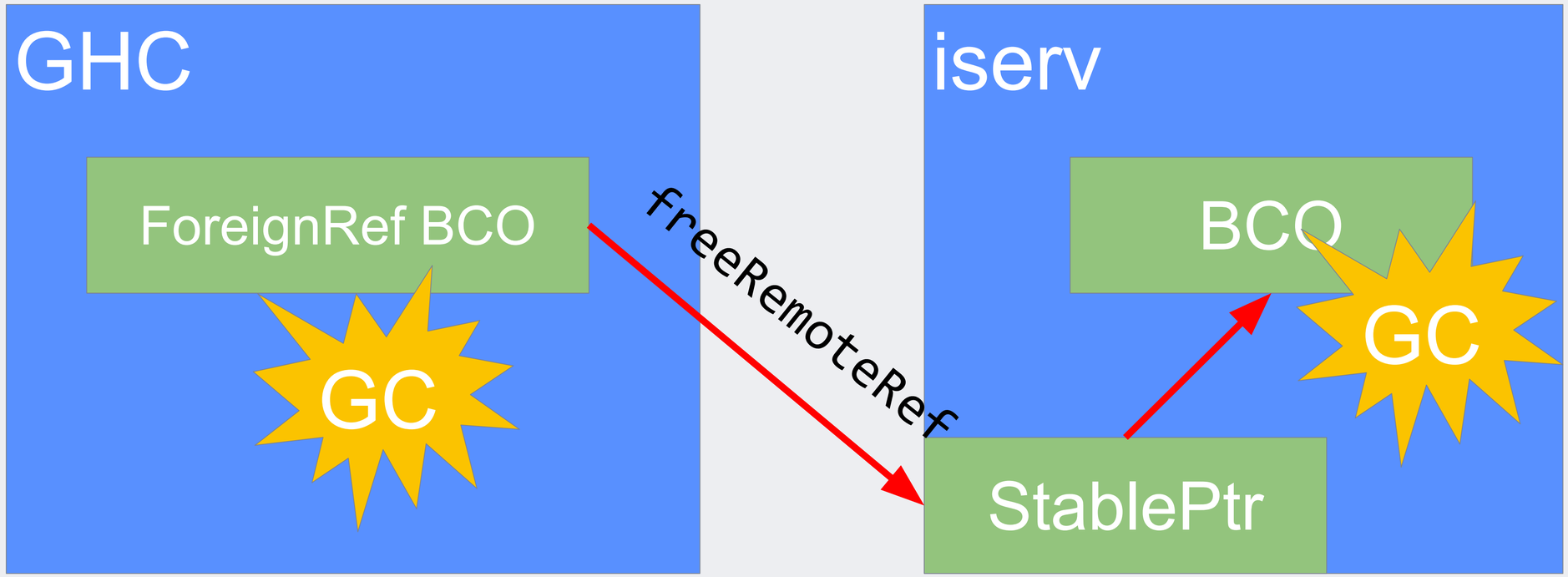
```
data Message a where
  -- | Create a set of BCO objects, and return HValueRefs to them
  CreateBCOs :: [LB.ByteString] -> Message [HValueRef]
  ...

iservCmd :: Binary a => HscEnv -> Message a -> IO a
```

Garbage-collected remote references

- Encode `Message` using `binary`
  - (with Generic deriving)
-

ForeignRef

- GHC retains byte-code with finalizers & StablePtr



Performance?

- Small overhead on `:load`, mainly when linking
- `binary` is a bit slow, use something faster in the future.
- If you want a fast GHCi:
  - ghci -j8 +RTS -A64m -qb0
  - GHC will also parallelise some of the serialization
  - Usable with your project

What does it buy us?

1. Stack traces with `ghci -prof`
2. `TemplateHaskell` with `-prof` no longer needs two compilation steps (and is safer)
3. `TemplateHaskell` no longer implies `-dynamic-too`
4. GHCJS has something similar for TH
   - unify in the future
5. Lays the groundwork for TH & cross-compilation

What could it buy us later?

- Multiple GHCi sessions in a single GHC
- Could run TH without any runtime linker at all
  - By statically linking a ghci-iserv binary at compile-time and then running it

Back to stack traces

	Changes source?	Runtime overhead?	Recompile req'd?	Accurate stacks?
Profiling	NO	YES	YES (but in GHCi you recompile anyway)	YES
HasCallStack	YES (but you choose where to use it)	YES (low overhead if you don't use it much)	NO	YES (but lexical only)
DWARF	NO	NO	NO	NO

Will it be the default?

- I hope so.
- But we can’t remove in-process GHCi:
  - Some GHC API use cases need it (Haskell for Mac)
  - GHC plugins must run in-process

- Docs:
  http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html#running-the-interpreter-in-a-separate-process
- Design notes:
  https://ghc.haskell.org/trac/ghc/wiki/RemoteGHCi
- Implementation notes:
  https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/ExternalInterpreter