(have lambdamon-exe already open, careful to have a var with just over 200 health)

Intro

- Ok, so let's start
- big thanks to anyone involved with organizing this conference
- first conference I do not only attend virtually, but also physically
- like it very much so far
- Introduce yourself, who, what you do
- I'm gonna talk about ...

Motivation

- Much confusion in the internet community
- many options are obsolete
- different approaches to the 'plugin problem' have different tradeoffs

Problem description

- Before I show you what I mean by 'the plugin problem'
- introduce you to lambdamon
- small game I made as use case (in light of the recent pokemon go hype)
- it is what happens if you mash together pokemon and the lambda calculus into a terminal UI game

explain concepts:

1. Principle: Capture all the free vars with my binderballs!
2. Problem: Aren't easy to capture with full health, need to beta reduce first
3. Success! 2 points
4. Use rename on next, kill/capture it, procede
5. staring at those free vars takes its toll, concentration too low, drink coffee

- have a background in 'extending game clients by an artifical intellegence' a.k.a botting
- would like other people to provide AIs for the game
- that's where plugin architecture comes into play
- show the types (Plugin, GameState, Move)
- so with the functionality pinned down that our plugins should provide...

Plugin Architecture Requirements

- broadly speaking:
  - Enable third parties to extend my game.

    - Clients, which wants to make use of their plugins, should not have to install a compiler toolchain
- more concretely, I came up with 5 criteria I'd look out for in a plugin architecture
- [Extensibility]
    - so the client can extend the application with third party code just by putting the right files in the right places
    - multiple plugins should be able to coexist
    - [Haskell] as extension language
- [Stand-alone]
    - minimal dependency footprint
    - should not require a compiler toolchain to be installed on the client machine
    - should just work in a fresh VM
- [Type safety]
    - incompatible extensions are recognized early and gracefully (not leading to crashes at runtime)
    - the plugins may even announcing supported versions
- [Maturity]
    - compiles, no showstopping bugs
    - easy integration with stack and cabal based plugin builds
    - ideally featuring a nice API

Shootout

- with those requirements in mind, let's procede with the shootout

Contenders

- sorry about using a Venn diagram here, but I thought it fit pretty well
- 3 different approaches:
    - blue Embedding a scripting language: We have hslua and hint as proxies here
    - red configuration through dynamic recompilation: check upon program start for changed source files - yi, xmonad and dyre
    - green hot code loading: Use the GHC API to link in the compiled object files into the running process - dynamic-loader and plugins

hslua

embed lua interpreter

- show code
- everyone who knows the C API feels at home;
- hslua is essentially a thin layer over the FFI, getting rid of some nasty details in error handling
- main takeaway:

    - script file
    - return an anonymous function
    - taking 4 arguments for the game state
    - returning the index of the chosen move
- implement simple.lua (health, damageMultiplier, concentration, hasCoffee)
- back to slides
- [Extensibility]
    - just drop in your files
    - hugely successful in areas like game programming (think world of warcraft)
- [haskell]
    - though lua is not haskell
- [stand-alone]
    - practically no dep footprint
    - everything statically linked
- [safety]
    - not type safe at all
    - lua in itself is not
    - API boundaries aren't checked either
    - lua stack is really easy to mess up, too low-level, takes much trial and error/tests
- [maturity]
    - as mature as it gets
    - probably seen more use than anything in the entire Haskell ecosystem
    - although hslua is rather low-level

hint

embed haskell interpreter

show code

- API much higher-level than hslua's
- mirrors GHCi, think GHCi-API
- just load module, settoplevelmodule for interpreter session and execute a string
- EXECUTE A STRING :(
- show ../lambdamon-hint/Plugin.hs
- some pretty complex business logic!
- can use extensions
- relies on the GHC package db
- GHC_PACKAGE_PATH hackery
- going through stack

back to slides

- [Extensibility]

    - just drop stuff in
    - tricky if you have more plugins with dependencies, because GHC pkg-db
- [haskell]
    - write stuff in Haskell, yay!
- [stand-alone]
    - not easily deployable to another machine because of
    - the pgk-db
    - hardcoded machine-specific paths
- [safety]
    - read/show serialization sucks
    - non-showable stuff
    - interpret + Typeable didn't work (maybe I did something wrong). Possibly unsafeCoerce always coerced the output to Reduce (0)
- [maturity]
    - many uses
    - generally nice API after you figured out the parallels to GHCi

dyre

- dynamic reconfiguration through recompilation (dyre), approach taken by yi and xmonad
- haven't really got it to work in the way I wanted, relies on GHC resp. stack ghc which is insufficient for any real-world use
- have tried to get it to work with yi, but found it far too complicated compared to just compiling stuff yourself when you changed your config
- although I want to show you the general approach
- dyre wraps around your program entrypoint
- upon program start looks for source file changes at the config path
- will recompile as necessary and call your entrpyoint with the appropriate config

next slide

- [Extensibility]
    - can't have more than one config file, merging configs requires knowledge of haskell
- [haskell]
- [stand-alone]
    - needs the whole compiler toolchain to be available
- [safety]
    - no api boundary, single type-checked and compiled program
- [maturity]
    - nicely documented, though mind-bending approach
    - relies on GHC and the global package registry
    - last package upload in 2014

dynamic-loader

- hot code loading
- linking in compile GHC object files at runtime

show code

- pretty thin layer over GHC API
- load actual objects from package archive *.a
- load qualified functions, highly unsafe
- pretty low-level, imperative interface...
- but it works
- only part of the story
- strip off symbol prefixes introduced by GHC
- vary based on GHC versions
- in GHC 8 will be the whole installed package identifier
- resulted in a really ugly makefile, with some really ugly and brittle regex matching
- Still looking into a reliable and easy build process
- ezyang recommended to patch the GHC RTS for that kind of functionality

back to slides

- [Extensibility]
  - third parties have an easy life just dropping in the correct archive file
- [haskell]
- [stand-alone]
  - Really stand-alone, although depending on GHC API
- [safety]
  - symbol prefixes depending on the installed package identifier get in our way
  - really needs reproducible builds, otherwise the object loader complains
  - Type errors at API boundaries result in crashes.
- [maturity]
  - unwieldy, scarcely documented API. Handling GHC generated symbols is an open problem. no usages

plugins

- also hot code loading, was the first to do so
- has a much nicer API than dynamic-loader, but has gone obsolete and is completely broken by now (as evidenced by multiple sources)
- even has support for custom GHC package databases, so a plugins could specify it's own local package database for its package dependencies

summary

- to conclude, here is a nice table summing up the shootout
- no winners, most are wounded, plugins is even bleeding out it seems
- will publish a more elaborate blog post/wiki entry soon(tm)

personally

- I'd really like hot code loading to work (seems like the most professional choice), but it's far too brittle, ABI not stable between GHC versions, installed package id, etc.
- dyre/xmonad/yi's approach is not worth the trouble (think about development, profiling, optimization, debugging, ghc flags...)
- stick to just compiling with the new config yourself or use a proper config file format/language
- I like the ease of using lua, but with a typed language like haskell.
- Might be interesting to implement
- another item for my ToDo list

last slide

- So that's it from me, guys, thanks
- I tried hard to make it so that you don't fall asleep, I partially succeeded I see