Plugin Architectures in Haskell

An Overview over the ecosystem

Sebastian Graf
September 15, 2016
https://github.com/sgraf812/hal16
Motivation
Problem Description

\[(\forall x xx)(\forall x xx)\]
Plugin Architecture Requirements

- Extensibility through third-party code
- Haskell as extension language
- Stand-alone: No compiler toolchain should be required on the client
- Type safety: Early and graceful recognition of incompatible extensions
- Maturity: Easy integration in a Cabal build process
Extensibility through third party code
Extensibility through third party code

Haskell as extension language
Plugin Architecture Requirements

**Extensibility** through third party code

**Haskell** as extension language

**Stand-alone** No compiler toolchain should be required on the client
Plugin Architecture Requirements

**Extensibility** through third party code

**Haskell** as extension language

**Stand-alone** No compiler toolchain should be required on the client

**Type safety** Early and graceful recognition of incompatible extensions
Plugin Architecture Requirements

Extensibility through third party code

Haskell as extension language

Stand-alone No compiler toolchain should be required on the client

Type safety Early and graceful recognition of incompatible extensions

Maturity Easy integration in a Cabal build process
Shootout
Extensibility  can’t be easier for third parties, see WoW. ✔✔

Haskell is not lua. ✗

Stand-alone The C bits are statically linked, no further dependencies. ✔✔

Type safety Neither in called code nor at API boundaries, also lua stack. ✗✗

Maturity lua is battle-tested and dead simple to include, yet hslua’s API is rather low-level. ✔
**Extensibility** Just drop in source files, although package dependencies are resolved through GHC package registry ✓

**Haskell** ✓

**Stand-alone** Uses the GHC API, including compilation specific settings paths ✗

**Type safety** through broken `Typeable` overloads, falling back to `read/show` serialization. ✗

**Maturity** Most-used (52 reverse deps) contender according to `hackage`. ✓
Dyre.wrapMain :: (Config -> IO ()) -> IO ()
Main.realMain :: Config -> IO ()
Extensibility  You can’t have more than one config file. Merging them requires knowledge of Haskell. ✗

Haskell  ✔

Stand-alone  Needs a complete compiler/stack toolchain available. ✗✗

Type safety  There are no API boundaries, it’s all one program and consequently type-checked as one. ✔✔

Maturity  Rotting. Only really works with GHC and the global package registry. Mind-bending API. ✗
**dynamic-loader**

**Extensibility**  Just drop in object archives. ✔️

**Haskell**  ✔️

**Stand-alone**  Although it depends on the GHC API, it works on a fresh installation. ✔️

**Type safety**  Needs reproducible builds in order to work seamlessly. Installed package id needed to find objects. Type errors at API boundaries lead to crashes at runtime. ❌

**Maturity**  Unwieldy, scarcely documented API. Handling GHC generated symbols is low-level and unresolved. 0 reverse deps. ❌
A word about plugins

**Extensibility** Just drop in object files. Package dependencies via `package.conf` files (outdated) ✔✔

**Haskell** ✔

**Maturity** Nicer API than `dynamic-loader`, but it’s horribly outdated and broken. XXX
<table>
<thead>
<tr>
<th></th>
<th>hslua</th>
<th>hint</th>
<th>dyre</th>
<th>dynamic-loader</th>
<th>plugins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensibility</td>
<td>✔ ✔</td>
<td>✔</td>
<td>✗</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>Haskell</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Stand-alone</td>
<td>✔ ✔</td>
<td>✗</td>
<td>✗ ✗</td>
<td>✔</td>
<td>?</td>
</tr>
<tr>
<td>Type safety</td>
<td>✗ ✗</td>
<td>✗</td>
<td>✔ ✔</td>
<td>✗</td>
<td>?</td>
</tr>
<tr>
<td>Maturity</td>
<td>✔</td>
<td>✔</td>
<td>✗ ✗</td>
<td>✗</td>
<td>broken</td>
</tr>
</tbody>
</table>
Thanks! Questions?
References


Check out the code of this talk at https://github.com/sgraf812/hal16