The “new” GHC API

And an example client: Scion

Thomas Schilling
nominolo@googlemail.com

An example

```
import GHC
import GHC.Paths ( libdir )  -- package ghc-paths

main =
  defaultErrorHandler $
  defaultCleanupHandler $
  runGhc (Just libdir) $ do
    dflags <- getSessionDynFlags
    setSessionDynFlags (dflags{ hscTarget = HscNothing })
    t <- guessTarget “A.hs” Nothing
    setTargets [t]
    result <- load LoadAllTargets
    if (succeeded result) then putStrLn “Yay!”
                          else putStrLn “D’oh!”
```

GHC needs to know the path where GHC is installed so it can find the package database and other things. The most portable way to do this at the moment is to use the ghc-paths library which uses some installation-time preprocessor magic to figure this out. I don’t know how well that works for deployment, though.

The main function sets up some error handlers and starts a GHC session. GHC API functions actually work for any monad that provides sufficient features (more on this in a later), but a default monad “Ghc” and a default monad transformer “GhcT” is provided. “runGhc” performs most of the necessary initialisation steps. One remaining step is to set the DynFlags, if you don’t modify the flags Ghc will die on you (I might change that in the future).

GHC has both static and dynamic flags. The former cannot be changed during a session, in fact, they cannot be changed during the whole run of a program using the GHC API as they are implemented as global variables (yes, really). In the long term we’d like to get rid of them, but this may require quite substantial refactorings. The DynFlags on the other hand can be set both on the command line and, more importantly, in an OPTIONS_GHC pragma of a source file. In the example above we set the compilation target to “nothing” which means that no code is generated, only parsing and type checking.

The next step is to set a target. A target is a module that we are interested in. Before we can type check a target we have to type check and load all its dependencies though. The example code is essentially what GHCi does when you type :load A.hs. “guessTarget” tries to automatically detect whether the string refers to a file or a module name and constructs the corresponding result of type Target. You could do it manually, but guessTarget is mostly fine.

An example

```
import GHC
import GHC.Paths ( libdir )  -- package ghc-paths

main =
  defaultErrorHandler $
  defaultCleanupHandler $
  runGhc (Just libdir) $ do
    dflags <- getSessionDynFlags
    setSessionDynFlags (dflags{ hscTarget = HscNothing })
    t <- guessTarget "A.hs" Nothing
    setTargets [t]
    result <- load LoadAllTargets
    if (succeeded result) then putStrLn "Yay!"
                          else putStrLn "D'oh!"
```

(ExceptionMonad m, MonadIO m) => DynFlags -> m a -> m a

Maybe FilePath -> Ghc a -> IO a

GhcMonad m => m DynFlags

GhcMonad m => DynFlags -> m [PackageId]

GhcMonad m => String -> Maybe Phase -> m Target

GhcMonad m => LoadHowMuch -> m SuccessFlag

Finally, the call to load loads our target and all its dependencies and returns a flag whether it was successful. This is example is actually missing some error handling since load might fail and throw an exception in some cases. If load fails to compile a module it will print all warnings and errors to stdout by default, but you can override this behaviour by installing proper hooks.

The Ghc Monad (Class)

```
class (Functor m, MonadIO m, WarnLogMonad m, ExceptionMonad m)
    => GhcMonad m where
  getSession :: m HscEnv
  setSession :: HscEnv -> m ()

class Monad m => WarnLogMonad m where
  setWarnings :: WarningMessages -> m ()
  getWarnings :: m WarningMessages

class Monad m => ExceptionMonad m where
  gcatch :: Exception e => m a -> (e -> m a) -> m a
  gblock   :: m a -> m a  -- async exception stuff
  gunblock :: m a -> m a

newtype Ghc a = Ghc { unGhc :: Session -> IO a }
data Session = Session !(IORef HscEnv) !(IORef WarningMessages)
```

GHC API functions typically are of the form “GhcMonad m => arg1 -> arg2 -> m result”. This way it is easy to use them in custom monads without having to wrap them in “liftGhc” or similar things. A “GhcMonad” needs to be able to perform IO actions, accumulate warnings and handle exceptions. For this reason we have added the “ExceptionMonad” class. This function should really be moved into a standard library, but for now it is GHC-specific.

“WarnLogMonad” is a simple writer monad with a clearing operation. For simplicity (and flexibility) it uses a state interface as primitives. The default monad “Ghc” uses is implemented as reader monad with mutable refs in the hope that this eliminates some opportunities for space leaks. It also makes it easier to implement “gcatch” and embedding the monad into IO for use in IO callbacks (needed by GHCi for editline callbacks, see “HscTypes.reflectGhc” and “HscTypes.reifyGhc” in the GHC API haddock docs.)

DIY Compilation

Arguments: (1) excluded modules (2) allow duplicate roots?

```
depanal :: GhcMonad m => [ModuleName] -> Bool -> m ModuleGraph
parseModule :: GhcMonad m => ModSummary -> m ParsedModule
typecheckModule :: GhcMonad m => ParsedModule -> m TypecheckedModule
desugarModule :: GhcMonad m => TypecheckedModule -> m DesugaredModule
loadModule :: (TypecheckedMod mod, GhcMonad m) => mod -> m mod

class ParsedMod m where
  modSummary   :: m -> ModSummary
  parsedSource :: m -> ParsedSource

class ParsedMod m => TypecheckedMod m where
  renamedSource     :: m -> Maybe RenamedSource
  typecheckedSource :: m -> TypecheckedSource
  moduleInfo        :: m -> ModuleInfo
  tm_internals      :: m -> (TcGblEnv, ModDetails) -- ignore me

class TypecheckedMod m => DesugaredMod m where
  coreModule :: m -> ModGuts
```

While load is very smart in loading all dependencies (it even skips things that don't need recompilation), sometimes we need to touch all modules anyways (e.g., Haddock.) In those cases it is still quite easy to perform the necessary steps "manually".

"depanal" performs a dependency analysis with the current targets as root and returns a module graph as a result. A module graph consists of a "ModSummary" which is the module name, its imports, and some extra information like the timestamp of the last modification the cached preprocessed sources and some further things. The remaining functions form a simple pipeline, each adding to the information of the previous phase (look out for space leaks!).

The typeclasses shown are faking extensible records, i.e., a desugared module has all the information of a typechecked module and some more and a typechecked module has all the information of a parsed module and some more. These classes should generally be treated as private since it's quite hard to construct, say, a valid "TypecheckedSource" datatype.

"loadModule" makes sure that we can load dependent modules by generating an interface file (in memory or on disk or both).

DIY Compilation (cont'd)

Arguments: (1) drop .hi-boot files? (2) modules (3) optional root module
Returns: Strongly-connected components

```
do
  -- ...
  mg <- depanal [] False
  let mods = flattenSCCs (topSortModuleGraph False ms Nothing)
  forM mods $ \modsum -> do
    tcm <- loadModule =<< typecheckModule =<< parseModule modsum
    printWarnings  -- also clears accumulated warnings
    -- ... extract info some info from tcm
    return (Just 42)
   `gcatch` \(e :: SourceError) -> do
              printExceptionAndWarnings e  -- clear warnings, too
              return Nothing
```

If we want to load all modules we must traverse them in dependency order. The result of "depanal" is not guaranteed to be in dependency order. "topSortModuleGraph" calculates the strongly connected components (SCC) which we can then flatten to get a dependency ordered module list. Note how the first argument to "topSortModuleGraph" is "False" which means, that we keep "hs-boot" files, which are used to break recursive dependencies in GHC. If you don't do this, then the result might be cyclic and compilation will fail. If, however, you only want to draw a module dependency graph then passing "True" is probably what you want.

Each of the "*Module" functions may encounter an error. In order to allow the simple pipelining structure, we throw these as an exception. If GHC encounters, say, a type error it doesn't give up immediately, though, so a "SourceError" may contain multiple error messages (or none if we failed due to -Werror). Warnings are accumulated for all the phases. In fact, even "loadModule" may generate warnings (orphan warnings). These are logged using the mechanisms of the GhcMonad, and can be extracted using "getWarnings" and cleared using "clearWarnigs". "printWarnings" does both for us and prints them to stdout. Similarly "printExceptionAndWarnigs". Remember, though, if you use custom error handlers you need to clear warnings yourself.

Error Handling

- Most functions operating on source files throw a "SourceError" if something goes wrong.

```
data SourceError = SourceError ErrorMessages
```

- GhcException (red = needs rethinking, arguably a SourceError)

```
data GhcException
  = PhaseFailed String ExitCode  -- an external phase (eg. cpp) failed
  | Interrupted                  -- someone pressed ^C
  | UsageError String    -- prints the short usage msg after the error
  | CmdLineError String  -- cmdline prob, but doesn't print usage
  | Panic String         -- the `impossible' happened
  | InstallationError String     -- an installation problem
  | ProgramError String          -- error in the user's code, probably
```

- There's also an "ApiError" exception, which indicates misuse of the API, however, it is rarely used ATM and may get merged into

A "SourceError" represents an "error in the source code", i.e., nothing external. GHC also has a custom exception type which partly overlaps with the definition of SourceError, so these things may change. Another change, I'd like to make is to have a datatype with each type of error message as a different constructor. That way it's easier for tools to recognise the error and suggest default fixes (instead of grepping through the message text).

The GHC API still has a couple of non-trivial pre-conditions for certain functions. Many will simply lead to a panic, a very small number has already been replaced by a "ApiError", more may follow. (OTOH, an API error is rarely recoverable.)

Compilation Modes and Targets

```
data GhcMode
  = CompManager          -- --make, GHCi, etc.
  | OneShot              -- ghc -c Foo.hs
  | MkDepend             -- ghc -M

data HscTarget
  = HscC                 -- -fvia-C
  | HscAsm               -- native code generator
  | HscJava              -- dead
  | HscInterpreted       -- generate bytecode
  | HscNothing           -- don't generate anything

data GhcLink
  = NoLink               -- Don't link at all
  | LinkBinary           -- Link object code into a binary
  | LinkInMemory         -- Use the in-memory dynamic linker
  | LinkDynLib           -- Link objects into a dynamic lib
                         -- (DLL on Windows, DSO on ELF platforms)
(underline = default, C/Asm depends on how GHC was compiled)
```

These variables (set in the DynFlags) control what output and kinds of code GHC produces. They should be fairly self-explanatory. Note that "LinkInMemory" works for all targets. Also note that not all code can be compiled to interpreted form, in particular unboxed tuples are not supported.

Extracting Information

```
isLoaded :: GhcMonad m => ModuleName -> m Bool
getBindings :: GhcMonad m => m [TyThing]   -- from GHCi session
data TyThing = AnId     Id
             | ADataCon DataCon
             | ATyCon   TyCon
             | AClass   Class
getModuleInfo :: GhcMonad m => Module -> m (Maybe ModuleInfo)
modInfoTyThings :: ModuleInfo -> [TyThing]
modInfoTopLevelScope :: ModuleInfo -> Maybe [Name]
modInfoExports :: ModuleInfo -> [Name]
modInfoInstances :: ModuleInfo -> [Instance]
modInfoModBreaks :: ModuleInfo -> ModBreaks
...
lookupGlobalName :: GhcMonad m => Name -> m (Maybe TyThing)
lookupName :: GhcMonad m => Name -> m (Maybe TyThing)
...
```

- ghc-syb package provides SYB instances for many GHC types

There are already lots of functions to query GHC's internal state and compiled modules. For a full list see the Haddock documentation. Some things are still missing or could use a better interface, but it's quite hard to figure out how things can be improved without knowing the client. So, if you have an idea for a client and feel like an important feature is missing – talk to us!

Issues

- The GHC API is not thread-safe(!)
- Inconsistent error types
- It's hard to hide certain internals. Clients must make sure not to break certain invariants.
- Documentation (We have Haddocks, though still very incomplete)
- Binary compatibility problems bite us hard: A GHC API client can only read .hi files that were written by the same (minor) version of GHC (or by the API client itself.)
- Flat module hierarchy. This will change after 6.10.2 has been released.

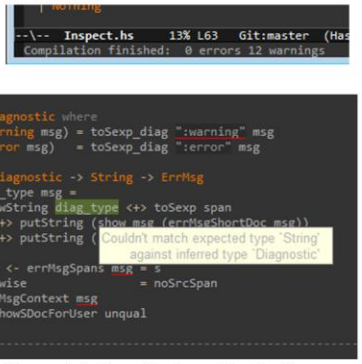
Sadly, the GHC API is not thread safe. It uses IORefs all over the place – e.g., for global variables and some caches. Interface files are also loaded lazily so there's a bit of unsafeLaunchMissile stuff.

GHC's AST types are quite huge and come with some invariants which make them quite difficult to use. It may be possible to translate to, say, haskell-src-exts ASTs but we should not expect such thing to go both ways. In addition, it might get changed with every version of GHC. We don't really know what a good solution would be.

Documentation status is improving slowly. Of course, help is always welcome. Once 6.10.2 has been released GHC will get a more standard hierarchical module structure which should also help with accessibility. (We wait until 6.10.2 because otherwise backporting could become a nightmare.)

Scion (rhymes with "lion")

- Scion is an IDE **library**
- So far:
  - open a Cabal library/executable
  - load it
  - re-typecheck current module
- In progress
  - lookup thing at point *
  - type of expr/id/pattern at point *
- Planned
  - Typecheck only parts of a module
  - Hoogle integration
  - GHCi integration (e.g. run selection)
  - Goto definition (cross package)
  - who-calls / who-uses (xrefs)
  - (type-directed / fuzzy) completion
  - Type of hole
  - expand cases
  - ...



(Screenshots from the Emacs frontend)

http://github.com/nominolo/scion
http://code.google.com/p/scion-lib

* only if whole module typechecks

There will never be a single definite Haskell IDE – many newcomers know and like Eclipse or Visual Studio, real (or wannabe) hackers use Emacs or Vim, or perhaps someday Yi, furthermore educators may want to provide special learning frameworks.

The goal is to put functionality that IDEs (or other tools working on Haskell code) can use into one place. The frontend should then merely wrap scion functionality. Existing code is integrated where possible (Visual Haskell, Shim).

Scion currently uses Cabal to set flags, but Cabal doesn't have a good API, either. Scion should not require a custom project file format, but provide an API to fill in settings from the frontend's project description.

The Emacs frontend is currently part of the repo because I use it. It communicates via a local socket with a server written in Haskell which only provides a small wrapper around the actual Scion API. (Most of the client side code has been adopted from SLIME.) Other non-Haskell clients could use the same strategy.

I also plan to add a little bit of concurrency despite GHC API's single-threadedness. Operations like querying the package database or supported language flags can be done in parallel to compilation even though we cannot compile or typecheck in parallel.