Using **Hugs** as a "Haskell server" (Draft)

Alastair Reid
Yale University
Department of Computer Science
New Haven, CT 06520
reid-alastair@cs.yale.edu

8 April, 1997

1 Introduction

[Warning: the Haskell server is still under development - you should expect to see changes in the server API from one release of **Hugs** to the next.]

Hugs is normally used as an interactive program. However, there are situations in which you want to use **Hugs** as a non-interactive system. Examples include:

- writing shell scripts in Haskell
- writing cgi scripts in Haskell
- writing Netscape plugins to let you embed Haskell code in HTML documents (the same way that you might use Javascript or Java)

For these purposes, we provide a "**Hugs** Server API" which provides access to some of **Hugs**' innards:

- loading/compiling files and projects
- compiling expressions
- constructing and evaluating "Graphs"

This is not enough to implement the **Hugs** user interface, but it's good enough for all the applications listed above. (We've done all three.)

2 Example

Here's a complete example of how to use the **Hugs** server. This is a simplified version of the "runhugs" program which loads a file, executes `Main.main` and returns the resulting exit code. (We've left out all error handling to keep things simple in this version.)

```
 1> #include "server.h"
 2> extern HugsServerAPI* initHugsServer Args((int,char**));
 3>
 4> static char* hugs_argv[] = {
 5>  "runhugs",   /* program name */
 6>  "+l"         /* literate scripts as default */
 7> };
 8> static int hugs_argc = sizeof hugs_argv / sizeof hugs_argv[0];
 9>
10> main( int argc, char** argv)
11> {
12>   HugsServerAPI* hugs = initHugsServer(hugs_argc,hugs_argv);
13>   hugs->setOutputEnable(0);
14>   argc--; argv++;
15>   hugs->setHugsArgs(argc,argv);
16>   hugs->loadFile(argv[0]);
17>   hugs->lookupName("Main","main");
18>   exit(hugs->doIO());
19> }
```

Here's what each line does:

1-2 Include the server API (included in appendix A)

4-8 Declare command line arguments used when initialising the server. These should consist of the program name (`argv[0]`) and a list of flags. Unlike **Hugs** you should not include files or projects to load.

12 Initialise the server. This returns a "virtual function table" which is used to access all other functions in the server API. (This is described in section 3.)

13 Turn off output from the compiler. This does not affect output produced by running Haskell code.

14 Forget the first argument on the command line. On a Unix system, this will be the name of the above C program.

15 Set the values seen by the Haskell functions `System.getProgName` and `System.getArgs`.

16 Load and compile the file named on the command line.

17-18 Lookup the Haskell function `Main.main` (which should be defined in the file we just loaded and should have type `IO ()`). The value returned is used as an exit code.

3 Initialising the server

The "Hugs server" is initialised by calling `initHugsServer`

```
> HugsServerAPI* initHugsServer(
>   Int    argc,
>   String argv[]   /* command line flags (-P, etc) */
>   );
```

This loads the standard Prelude and the dynamic typing library (see section 8) and processes any command line flags in argv.

If initialisation succeeds, it returns a "virtual function table" containing all the other server functions you can call. That is it returns a non-null pointer to a struct of type `HugsServerAPI`. We'll go through these in detail in the rest of the document — but here's the complete list:

```
> typedef struct _HugsServerAPI {
>     char* (*clearError     ) (void);
>     void  (*setHugsArgs    ) (int, char**);
>     int   (*getNumScripts  ) (void);
>     void  (*reset          ) (int);
>     void  (*setOutputEnable) (unsigned);
>     void  (*changeDir      ) (char*);
>     void  (*loadProject    ) (char*);
>     void  (*loadFile       ) (char*);
>     HVal  (*compileExpr    ) (char*,char*);
>
>     void  (*lookupName     ) (char*,char*); /* push values onto stack*/
>     void  (*mkInt          ) (int);
>     void  (*mkString       ) (char*);
>
>     void  (*apply          ) (void);      /* manipulate top of stack */
>
>     int   (*evalInt        ) (void);      /* evaluate top of stack   */
>     char* (*evalString     ) (void);
>     int   (*doIO           ) (void);
>
>     HVal  (*popHVal        ) (void);      /* pop stack               */
>     void  (*pushHVal       ) (HVal);      /* push back onto stack    */
>     void  (*freeHVal       ) (HVal);
> } HugsServerAPI;
```

In the rest of this document, we'll assume that you've put a pointer to the "virtual function table" in a variable called `hugs` and we'll write things like this

```
> void  hugs->loadFile    (char*);
> void  hugs->loadProject (char*);
```

to indicate the type of `hugs->loadFile` and `hugs->loadProject`.

4 Loading files

Loading files is easy enough. Simply call `hugs->loadFile(<name>)` or `hugs->loadProject(<name>)`.

```
> void  hugs->loadFile    (char*);
> void  hugs->loadProject (char*);
```

Some programs need to be able to "unload" (or "forget") some of the Haskell files that have been loaded. **Hugs** maintains a "stack" of all files it has loaded. To unload some files, it pops files off the stack. The server API provides two functions for modifying the stack of files: `getNumScripts` tells you how large the stack is; and `reset` sets the stack to the required size.

```
> int   hugs->getNumScripts (void);
> void  hugs->reset         (int);
```

Typically, one writes code like this to load and execute functions from a sequence of files. Note that the standard Prelude and the module `MyLibraries` is only loaded once.

```
> HugsServerAPI* hugs = initHugsServer(hugs_argc,hugs_argv);
> hugs->loadFile("MyLibraries");
> int baseLevel = hugs->getNumScripts();
> for(int i = 1; i < argc; ++i) {
>   hugs->reset(baseLevel);
>   hugs->loadFile(argv[i]);
>   hugs->lookupName("Main","main");
>   hugs->doIO();
> }
```

5 Executing Expressions

In section 2 we used `lookupName` to lookup `"Main.main"` and `doIO` to execute it. As you've probably guessed, `lookupName` leaves a "pointer" to `Main.main` on the stack and `doIO` evaluates the object found on top of the stack. Here are some of the other operations which operate on the stack:

```
> void  hugs->mkInt       (int);
> int   hugs->evalInt     (void);
>
> void  hugs->mkString    (char*);
> char* hugs->evalString  (void);
>
> void  hugs->apply       (void);
>
> void  hugs->lookupName  (char*,char*);
> int   hugs->doIO        (void);
```

The new functions are as follows:

- `mkInt` pushes (a representation of) an `int` onto the stack. `evalInt` evaluates the `Int` on top of the stack.
- Similarily, `mkString` pushes (a representation of) a C string onto the stack and `evalString` evaluates the `String` on top of the stack.
- `apply` pops an argument and a function off the stack (in that order) and applies the function to the argument. A typical usage is

  ```
  > hugs->lookupName("Foo","ackerman");
  > hugs->mkInt(4);
  > hugs->apply();
  > hugs->mkInt(2);
  > hugs->apply();
  ```

  Alternatively, you might define this macro

  ```
  > #define ap(f,x) f; x; hugs->apply();
  ```

  and write this

  ```
  > ap(ap( hugs->lookupName("Foo","factorial")
  >      , hugs->mkInt(4))
  >      , hugs->mkInt(2));
  ```

ToDo: *The server API currently provides no way to push floats, chars, etc onto the stack. There's no real problem in adding this, but we haven't needed it yet.*

6 Haskell Values

It's sometimes useful to be able to store the result of a calculation for later use. These operations allow you to pop Haskell Values off the stack, store them and later push them back onto the stack.

```
> HVal  hugs->popHVal     (void);
> void  hugs->pushHVal    (HVal);
> void  hugs->freeHVal    (HVal);
```

"Haskell Values" remain valid if you load additional Haskell files and if you evaluate expressions but are invalidated by calling `reset`.

Warning: *No check is performed to detect the use of invalid values; the result is likely to be messy.*

7 Compiling Expressions

The functions described in section 5 let you evaluate almost any Haskell expression but are rather painful to use. This version of the server provides a much more convenient function which lets you compile arbitrary Haskell expressions.

```
> HVal  hugs->compileExpr (char*,char*);
```

The function `compileExpr` takes two arguments. The first argument is the name of the module in which to evaluate the expression. The choice of module determines which functions are in scope. The second argument is the expression itself.

Portability Note: *The current version of the server includes the full* **Hugs** *compiler so that we can load the Prelude and other libraries. Since the compiler is included in the server, it is both cheap and easy to provide* `compileExpr`*. In future versions of the server, we'd like to be able to load precompiled versions of the Prelude and libraries and omit most of the* **Hugs** *compiler. In such a system, we would also omit* `compileExpr` *since it is possible to do most of what* `compileExpr` *does using* `lookupName` *and* `apply`*.*

ToDo: `compileExpr` *really ought to leave its result on the stack.*

8 Dynamic Types

The evaluation mechanisms described above make it very easy to construct and attempt to evaluate ill-typed objects. To avert catastrophe, the server typechecks very function application. The mechanisms used to perform this typecheck are not as flexible as the Haskell type system for two reasons:

- Typechecking is restricted to a small set of base types and type constructors. If you need to use other types, you'll need to define new instances of the `Typeable` class. Use the instances in appendix B as examples of how to write your own instances.
- Typechecking is restricted to *monomorphic* values. Looking up a polymorphic function will always result in an error. There are two solutions:

– Add monomorphic instances of the functions to your code. For example, if you need to use `Prelude.length` at 3 different types, you might write a module containing these definitions

```
> length_Int :: [Int] -> Int
> length_Int = length
>
> length_Ints :: [[Int]] -> Int
> length_Ints = length
```

– Use `compileExpr` to lookup the values at different types

```
> HVal length_Int  = hugs->compileExpr("Prelude","length :: [Int] -> Int");
> HVal length_Ints = hugs->compileExpr("Prelude","length :: [[Int]] -> Int");
```

In practice, both are equally irritating.

ToDo: *If we remove* `compileExpr` *we should probably improve the dynamic typing.*

9 Handling Errors

So far, we have assumed that errors almost never occur. In practice error-free execution is the norm: the standard prelude can't be found; filenames are wrong; programs contain syntax and type errors; modules don't define what they're supposed to; people look up polymorphic functions; Haskell code returns errors; etc.

The **Hugs** server is fairly robust: it tries to catch any errors and will not perform any further actions until the error is resolved. The function `clearError` is used to detect whether an error has occurred (since the last time `clearError` was called); to obtain any compiler output associated with the error; and to reset an "error flag".

```
> char* hugs->clearError (void);
```

All other functions in the server API return immediately if the error flag is set — this encourages programmers to call `clearError` frequently and prevents the server from being totally corrupted if `clearError` is not used.

The output returned by `clearError` depends on whether or not compiler output has been redirected to a buffer using the function `setOutputEnable`

```
> void hugs->setOutputEnable (unsigned);
```

If compiler output has not been redirected, `clearError` produces a brief error message. If compiler output has not been redirected, then `clearError` produces an error message followed by all the output that has been collected since the last time `clearError` was called.

Using these features, it's possible to write a more robust version of the runhugs program given in section 2.

```
> static void check() {
>   char* err = hugs->clearError();
>   if (err) {
>     fprintf(stderr,"Hugs Error:\n%s\n",err);
>     fflush(stderr);
>     exit(1);
>   }
> }
>
> main( int argc, char** argv)
> {
>   int exitCode;
>   HugsServerAPI* hugs = initHugsServer(hugs_argc,hugs_argv);
>   if (NULL == hugs) {
>     fprintf(stderr,"Unable to initialise Hugs\n");
>     fflush(stderr);
>     exit(1);
>   }
>   hugs->setOutputEnable(0);
>   check();
>   argc--; argv++;
>   hugs->setHugsArgs(argc,argv);
>   if (argc < 1) {
>     fprintf(stderr,"hugs standalone requires at least one argument\n");
>     fflush(stderr);
>     exit(1);
>   }
>   hugs->loadFile(argv[0]);
>   check();
>   hugs->lookupName("Main","main");
>   exitCode = hugs->doIO();
>   check();
>   exit(exitCode);
> }
```

A server.h

This is the current contents of the file `server.h`. This is the only file you need to include into programs that use the server.

```
/* --------------------------------------------------------------------------
 * Definition of the Hugs server API
 *
 * Copyright (c) The University of Nottingham and Yale University, 1994-1997.
```

```
 * All rights reserved. See NOTICE for details and conditions of use etc...
 * Hugs version 1.4, April 1997
 * ------------------------------------------------------------------------*/

#ifndef Args
# if HAVE_PROTOTYPES
#  define Args(x) x
# else
#  define Args(x) ()
# endif
#endif /* !defined Args */

typedef int HVal;     /* Haskell values are represented by stable pointers */

typedef struct _HugsServerAPI {
    char* (*clearError     ) Args((void));
    void  (*setHugsArgs    ) Args((int, char**));
    int   (*getNumScripts  ) Args((void));
    void  (*reset          ) Args((int));
    void  (*setOutputEnable) Args((unsigned));
    void  (*changeDir      ) Args((char*));
    void  (*loadProject    ) Args((char*));
    void  (*loadFile       ) Args((char*));
    HVal  (*compileExpr    ) Args((char*,char*));

    void  (*lookupName     ) Args((char*,char*)); /* push values onto stack*/
    void  (*mkInt          ) Args((int));
    void  (*mkString       ) Args((char*));

    void  (*apply          ) Args((void));      /* manipulate top of stack */

    int   (*evalInt        ) Args((void));      /* evaluate top of stack   */
    char* (*evalString     ) Args((void));
    int   (*doIO           ) Args((void));

    HVal  (*popHVal        ) Args((void));      /* pop stack               */
    void  (*pushHVal       ) Args((HVal));      /* push back onto stack    */
    void  (*freeHVal       ) Args((HVal));
} HugsServerAPI;

/* type of "initHugsServer" function */
typedef HugsServerAPI *(*HugsServerInitFun) Args((int, char**));

/* ------------------------------------------------------------------------*/
```

B The Dynamic module

```
module Dynamic
   ( Typeable(typeOf),
   , Dynamic, toDynamic, fromDynamic, dynApply,
   , fromDyn, dynApp,
   , intToDyn, fromDynInt, strToDyn, fromDynStr,
   , Tycon(..), Type(..)
   ) where

-----------------------------------------------------------------
-- Dynamics
-----------------------------------------------------------------

data Dynamic = ...

-- The core functions
toDynamic   :: Typeable a => a -> Dynamic
fromDynamic :: Typeable a => Dynamic -> Maybe a
dynApply    :: Dynamic -> Dynamic -> Maybe Dynamic

-- special cases
fromDyn     :: Typeable a => Dynamic -> a
intToDyn    :: Int    -> Dynamic
strToDyn    :: String -> Dynamic
fromDynInt  :: Dynamic -> Int
fromDynStr  :: Dynamic -> String
runDyn      :: Dynamic -> IO ()
dynApp      :: Dynamic -> Dynamic -> Dynamic

-----------------------------------------------------------------
-- Types
-----------------------------------------------------------------

data Tycon = Tycon String     deriving Eq
data Type  = App Tycon [Type] deriving Eq

unitTC    = Tycon "()"
intTC     = Tycon "Int"
integerTC = Tycon "Integer"
floatTC   = Tycon "Float"
doubleTC  = Tycon "Double"
charTC    = Tycon "Char"
ioTC      = Tycon "IO"
funTC     = Tycon "->"
```

```
listTC    = Tycon "[]"
tup2TC    = Tycon "(,)"

class Typeable a where typeOf :: a -> Type

-- Constant Tycons are easy

instance Typeable ()      where typeOf x = App unitTC    []
instance Typeable Int     where typeOf x = App intTC     []
instance Typeable Integer where typeOf x = App integerTC []
instance Typeable Float   where typeOf x = App floatTC   []
instance Typeable Double  where typeOf x = App doubleTC  []
instance Typeable Char    where typeOf x = App charTC    []

-- Non-constant Tycons require sneakiness

instance Typeable a => Typeable (IO a) where
  typeOf m =
    case unsafePerformIO m of { r ->
    App ioTC  [typeOf r]
    }

instance (Typeable a, Typeable b) => Typeable (a -> b) where
  typeOf f =
    -- We use case to bind arg and result to avoid excess polymorphism
    case undefined of { arg ->
    case f arg     of { result ->
    App funTC [typeOf arg, typeOf result]
    }}

instance Typeable a => Typeable [a] where
  typeOf xs = App listTC [typeOf (head xs)]

instance (Typeable a, Typeable b) => Typeable (a,b) where
  typeOf p = App tup2TC [typeOf (fst p), typeOf (snd p)]
```