

GHC-Maple Interface 0.2 (Eden Extension)

6th July 2004

1 Authors

- (c) 2003 Rafael Martinez Torres <rmartine@fdi.ucm.es>
- (c) 2003 Ricardo Peña Mari <ricardo@sip.ucm.es>
- (c) 2000 Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
- (c) 2000 Hans-Wolfgang Loidl <hwloidl@cee.hw.ac.uk>

2 Location

These packages can be obtained from

GHC-Maple:
<http://www.risc.uni-linz.ac.at/software/ghc-maple>

3 Description

This is an implementation of an interface between:

- Glasgow Haskell Compiler (<http://www.haskell.org/ghc>) ghc-5.02-3
- Maple (<http://www.maplesoft.com>) maple 7

Optionally, an extension for the interface is provided for:

- Marburg Eden Compiler (<http://www.mathematik.uni-marburg.de/~eden>)
mec-5.02-3 (release date at around 2003-02-12)

The usage of this interface is described in the sections "Program Initialization" and "Program Interface" below.

WARNING: I cannot assure the interface to run properly with other other versions of above products. In fact, some incompatibility is known when using Maple V, and ghc-6.0.

4 ChangeLog

4.1 Version 0.1 (2000):

Initial release. This version is constraint by two facts:

- Maple V does not export its kernel API, i.e, you can only use it as an end-user application. To arrange this, a client-server approach is designed in order to make interoperable both systems, implementing the protocol HM (Haskell-Maple). The Haskell program will act as a client and Maple as a server, each other communicating via two UNIX pipes connecting both respective standard input/output. “maple2.hs,maple.c” implement the client side, and “start.maple” implements the front end for the services exported by Maple.
- GHC-4.08.2 is not shipped with a POSIX library, so to get access for the O.S services (fork(),pipe()), the programmer must “hardwire” the code in C, and integrating it with Haskell via CCall interface (a deprecated ancestor of future FFI). Also a patch is scratched to allow synchronizing on a concurrent environment.

4.2 Version 0.2 (2003):

This version takes advantage of new futures of ghc-5.02.3 series, avoiding the last handicap of those listed above. However, essentially the underlying client-server strategy remains the same, since Maple 7 remains closed for development issues by third parts. Summing up:

4.2.1 Rewriting with POSIX library.

The POSIX library shipped with ghc-5.02.3 allows you to access the O.S services in a entirely functional style, i.e, you can create a pipe, to fork a process, make it to load and run a new binary image, but you must be carefull with the semantics of POSIX calls inside a STG-Haskell context.

More accurately: the data type to read from a a pipe, Fd -file descriptor, a data type out of the Haskell IO() subsystem-, and its associated function call,

```
fdRead :: Fd -> ByteCount -> IO(String,ByteCount)
```

may block the entire STG, (hence the rest ot the Concurrent threads) when a trying to read from the pipe. The proposed solution is to “promote” the type Fd into that specific of Haskell IO(), -Handle-, thanks to the function

```
fdToHandle:: Fd -> Handle
```

The GHC RTS frees you of dealing with non-blocking readings and writings thanks to:

```
hPutStrLn :: Handle -> String -> IO()
hGetLine  :: Handle -> IO(String)
```

Additionally, since the HM protocol is “Line-Oriented”, we can take advantage of the automatic flushing on Handles, by specifying on initialization:

```
hSetBuffering inpipe LineBuffering
hSetBuffering inpipe LineBuffering
```

4.2.2 Data Type for MapleObjects.

As consequence of the previous arrangement, the underlying data type for messages sending from/to Maple side is forced to be `-String-`, in the hope that this type will hold all values sent by Maple, specially for the internal format of objects, `(%a)`, since a `Byte` may not coincide with a `Char`.

Up to now, there is no empirical experience of failure, since all Maple encapsulated data types are “printable characters” (see `/tmp/debug`). If you take a look into “`start.maple`” (this code remain with no changes), the server send the message via a formatted string (`printf(“read: %a\n”,value)`).

So the things, data type for `MapleObject` has been simplified considerably: instead of `ByteArray`, you get

```
data MapleObject = MAR String
```

4.2.3 FFI (Foreign Function interface)

In order to make the HM protocol (pipes, sending, receiving) and the concurrency issues (`MVar`, see next point) transparent to the final user, we have to implement a sort of persistent state, forcing the “pure nature” of functional programming. We will use an “external context” entity (in the terms it is described on FFI manuals) written in C to record the pointers to Haskell objects we want to save and restore.

```
foreign import ccall crecordPointer :: StablePtr a -> IO()
foreign import ccall cgrabPointer  :: IO(StablePtr a)
```

In order to assure the pointed Haskell expression is not to be affected by the garbage collector, and alternatively, to grab the pointer and make it usable at Haskell land, you have to invoke prior and later the functions:

```
newStablePtr :: a -> IO (StablePtr a)
deRefStablePtr :: StablePtr a -> IO a
```

From the point view of the Type System, the monad `IO()` contributes to smooth this gap between the functional and imperative paradigms. You enter the same monad when recording/grabbing the pointers and when sending/receiving the streams. The only “malicious transgressor evil” (identified as “Mephisto in action”

in source code) does act when returning the result up to the caller, breaking the purity of the model, by using the well known “back-door” into the IO() monad (also used in 0.1)

```
unsafePerformIO :: IO a -> a
```

4.2.4 Making Maple Calls “THREAD_SAFE”

On version 0.1, the author sketched a transient mechanism to make concurrency available among Concurrent Haskell threads invoking Maple Interface functions. The mechanism was done via a sort of “active waiting” together with “yield()” invokes. Lacking the FFI implementation on ghc-4.08.2 series, author was not allowed to use a more appropriate abstract mechanism, like MVar (available from 3.02) , but he indicates the way it should be done. Now, thanks to the FFI interface mentioned above, we can use them in a “clean” way to make concurrency issues transparent to the calling user.

4.2.5 Eden extension: replicating Maple around the DREAM.

Please, go to 7

4.3 Version 1.X (?):

It’s known that MapleSoft policy has changed and the last released version Maple 9 exports the interpreter’s API in order to include Maple-Calls in your C program. A good idea in order to definitively reach an “steady version” of this interface would be write a Haskell-wrapper for this C-library via FFI, in such a way other famous Haskell libraries do, like HOpenGL. The final product would be more reliable, since all the “pipery, forkery” around the HM protocol would be entirely dropped. More properly, we are talking about a “Haskell binding” to Maple (as C, Java...) rather than an “interface”.

Note that exception handling on local Haskell side is not implemented. The new version would make it feasible in a easy way. Even more, I’m not sure, but if the API calls are “thread_safe”, we should not care to implement explicit synchronizing methods for Concurrent Haskell. All are advantages.

Concerning ELF issues, I’m not sure, but a final executable should be an aggregate of both libraries, libHSRts.a and libOpenMaple.so, coordinated by the FFI. This idea is to be matured.

5 Installation

For executing a Haskell (opt. Eden) program with this interface, you should set the environment variable MAPLEROOT to a directory of your choice, e.g.

```
(sh/bash) export MAPLEROOT=/home/rrmt/ghc-maple-0.2/mapleroot
(csh/tcsh) setenv MAPLEROOT /home/rrmt/ghc-maple-0.2/mapleroot
```

WARNING: CHECK always `MAPLEROOT` variable on every session you start, no matter if you are compiling or running. Debuggery is a weak point. So, a good way to include it is in the `.bashrc` file, or where you set the PVM environment var. In the case of Eden, not only the program, but even the `pvm3` daemon must hold it !!! The `MAPLEROOT` directory must contain the following files (after successful instalation):

- `start.maple` ... the program executed by maple
- `init.maple` ... a (possibly empty) file containing the application-specific initialization commands
- `libghcmaple.a`
- `MapleAPI.hi`

(TODO)This directory will also receive all internal error messages in a file `error.<pid>` where `<pid>` is the process id of the program execution.

5.1 + Installing Haskell version

```
bash-2.04$ tar xzvf ghc-maple-0.2.tar.gz
bash-2.04$ cd ghc-maple-0.2
bash-2.04$ export MAPLEROOT='pwd'/mapleroot
bash-2.04$ cd lib
-- You may have to adjust the Makefile flags.
bash-2.04$ make
bash-2.04$ make install
- Examples:
bash-2.04$ cd ghc-maple-0.2/examples
bash-2.04$ ls
bash-2.04$ cd fibonacci
-- You may have to adjust the Makefile flags.
-- Please, set the MAPLEROOT environment variable.
bash-2.04$ make
-- Note how you rewrite $MAPLEROOT/init.maple
bash-2.04$ nfibMaple 100
```

5.2 + Installing Eden version (optional)

Installing Eden is a bit more complex, since it requires a patch to allow Eden-process threads be able to read-write from a handle. Otherwise, your program will be interrupted:

```
rrmt@avila:/media/sda1/tmp/ghc-maple-0.2/examples/fibonacci> ./nfibMaple 8
```

```

==== Starting parallel execution on 1 processors ...
=nfibMaple: fatal error: blockThread: impossible why_blocked code 4 for TSO 3
libpvm [t40005]: pvm_upklong(): End of buffer
==== [40005] Uncontrolled termination at 40006 with exit(254)

```

Apply the patch to the Eden RTS following the next steps:

```

bash-2.04$ cd /path/to/mec-5.02.3/ghc/rts/parallel
bash-2.04$ patch-p0 < /path/to/HLComms.c.diff
bash-2.04$ cd ..
bash-2.04$ make

```

Note that you don't need to do "make install" since we use the compiler under "ghc-inplace". By the way, in the time this patch was applied we report implementors of MEC, namely Jost Berthold, a strange failure on closing Eden programs. Jost could not then reproduce the bug easily. I guess the previous patch may have to do some thing with this error. Fortunately it only happens at the end of the computation. If this happens, the only thing you have to do is reset the PVM.

```

==== Starting parallel execution on 1 processors ...
{createThread}Daq ghuH: refusing to create another thread; no more than 8192 threads al
==== [4000b] Termination at 4000c with exit(-1)

```

After that proceed as in the previous case for Haskell. Take in mind you must adjust the Makefiles in order to get a compilation for an Eden system. Basically you have to:

1. Choose appropriate compiler.
2. Uncomment the "-D_EDEN_" flag.

6 End user programs.

6.1 Program Interface.

The most important Haskell/Maple Interface functions are

```

string2MapleExpr :: String -> MapleObject
mapleObject2String :: MapleObject -> String
mapleEval :: String -> [MapleObject] -> MapleObject
mapleEvalN :: String -> [MapleObject] -> [MapleObject]

```

- string2MapleExpr converts a String denoting an expression in Maple syntax into a MapleObject, e.g.

```

- p = string2MapleExpr "x^2*y+x"

```

– v = string2MapleExpr "x"

- mapleObject2String converts a Maple Object into a string representation in Maple syntax, e.g.

– putStrLn (mapleObject2String p)

- mapleEval takes a string that denotes the name of a Maple function (builtin or defined by the initialization file "init.maple"), a list of Maple objects and applies the functions to the arguments. For instance, to call the Maple function "int(expr, x)", we may call:

– r = mapleEval "int" [p, v]

- mapleEvalN behaves similar to MapleEval but it may only be used on a Maple function that returns a (Maple) *list* of results.

6.2 Program Initialization.

The Haskell/Eden program must be always enclosed by calls of the corresponding functions. As an example, take this:

```
main :: IO ()
main =
do
#ifdef _EDEN_
a<- mapleInitAllDream
#else
  mapleInit
#endif
...
#ifdef _EDEN_
  b<-mapleTermAllDream
#else
  mapleTerm
#endif
```

At last, just a note for newbie people on functional programming. Due to underlying IO() lazy model on Haskell, you should always to force the “reduce normal form” of the Maple-dependent closure you were trying to evaluate before setting down the interface. Otherwise, the system will reduce to “weak head normal form”, and you may not get the expected result. Use “Strategies” library. If you don’t know what I’m talking about, just make Haskell to print your value before setting down the interface.

(This is not applicable to the “a”, “b” closures in the example above. They are programmed to do it internally).

6.3 Debuggery

Debuggery is certainly a weak point on this interface, since exception handling is not well implemented. So, if things go well, you can get a good trace. Otherwise, you may enter a race of failures. Most part of them have to do with inappropriate “MAPLEROOT” value, or “init.maple” format.

However, there are three points to have in mind if you want to trace the computation:

1. File /tmp/debug, where you can read all the trace output by the Maple server.
2. For a “verbosely” output of the interface, compile it with “-DDEBUG” flag.
3. 2004-06-15: After a lecture by Rita Loogen at Madrid, you will understand why messages on the error output seems to be cryptic: Having rewritten most part of C code in Haskell, most part of formatted strings send by Maple are misinterpreted, since Haskell cannot process them, i.e, you will certainly read something like:

```
error (%1 expects its %2 argument to be of type %3, but received %4)
```

7 Eden extension.

On 0.1 version there was an attempt to get a parallel version of this interface. This could be done in two ways: hacking the Maple side, so control of parallelism is transparent to Haskell programmers (unless required to deal expressions in Maple involving explicit parallelism), or trying to use one of the parallel dialects on the Haskell side: Gph, Eden...

This version implements Eden’s approach. As it is implemented as a sort of distributed Haskell RTS, with some minor arrangements on semantics at very special points and implementing some primitive functions to support “coordination model”, we decided to replicate also a process Maple for each PE (Processor Element) belonging to the DREAM machine.

Note also that Eden RTS is ready to support concurrent processes inside a PE, so those are intended to share the same Maple server, and you have to synchronize the use not only by different processes, but also from implicit threading run by Eden inside a process. That’s what we talk about in the next points:

7.1 Replicating Maple around the DREAM.

The immediate solution, i.e, setting up/down the interface on demand by a process, is discarded since it appears to be expensive and heavy in terms of time required by O.S services.

The strategy to follow would be rather the next: to setup a Maple server exactly once for each PE (1:1) and set down when it is no longer required. So the things, older versions used to set up/down the Maple server inside a process using the next trick: the first of these calls would effectively set up the interface, updating the initial value of a “persistent variable” from 0 into 1. The rest of processes trying to setup the interface, having read the “persistent-variable” as distinct from 0, will not setup anymore, but increase the value of the state-variable. Reverse operation is done when setting down the interface: only the last process holding a value of 0 for the state-variable will set down the interface, while prior processes are intended only to decrease such a value.

This solution becomes good in terms of efficiency, since Maple interface is only set up/down once per PE. However, a simple approach is feasible having in mind some of the implementation details of Eden RTS: the `Sysman.c`. This code is designed to spawn the distributed RTS on each PE prior to any computation by Eden. Our idea follows this approach: FROM EDEN SOURCE code, prior to any use of Maple interface, user have to call the function “`mapleInitAllDream`” in order to setup a Maple process for each PE. This is possible thanks to the constant “`noPE`” exported by Eden, and will result OK only when no prior Eden processes had been instantiated. Otherwise, the concurrency of two or more processes inside a PE could become a chaos inside a PE. The same holds for setting down the interface: only when no more processes remains instantiated we can warranty allocation 1:1 among processes and PE’s. Using this method, the rest of functions based on Maple Interface have not to care on the status of the Maple server.

```
mapleInitAllDream :: IO (Int) -- return how many PE's have been setup.
mapleInitAllDream =
  do
    let a = [process (mapleInitLocalDream) # (i) | i <- [1..noPe] ] 'using' spine
        return (length a) 'demanding' Strategies.rnf a
    mapleTermAllDream :: IO (Int) -- return how many PE's have been setup.
    mapleTermAllDream =
      do
        let a = [process (mapleTermLocalDream) # (i) | i <- [1..noPe] ] 'using' spine
            return (length a) 'demanding' Strategies.rnf a
```

- 2004-06-15: Would be nice if those implementing MEC compiler decide to incorporate a new “constant” on Eden API, holding the value of PE on which the processes is being run, i.e (`peID::Int`). That would contribute in the past functions to handle eventual errors on a concurrent allocation of processes/PE. Apart of this case of use, it seems to be interesting tool to distinguish future concurrency of processes inside a PE, and not too difficult to implement.

7.2 Synchronizing among implicit threads.

At last, we reach one of the “dirty” issues on the interface, due to the nature of underlying Eden execution model. From the very early times, Eden system was designed to support “implicit concurrency” inside a process, this means, in order to speed-up the evaluation of a process, the host RTS will instantiate one thread per element on the output tuple, not requiring a explicit call by the programmer.

What happens then if one of these elements involve some Maple function? Remember, that, under the single signature of the interface functions, a complex protocol (HM, pipes) is operating in a transparent way to the calling user: but, from the point of view of the interface, HOW CAN WE SYNCHRONIZE the concurrent access to the shared pipe by many (implicit) threads?

As Eden threading subsystem is implicit, Eden API does not export any explicit mechanism to support concurrency among implicit threads. Let’s to consider some of the next options:

7.2.1 QUEUEME closure.

The RTS implements the QUEUEME closure, a sort of synchronization node in the receiver’s heap linked to the receiving port in order to regulate concurrent access to a subgraph in normal form, but clearly this method does not fit our requirements.

7.2.2 Eden.Channame

In Eden, there is implemented a kind of explicit communicating channels you can pass as a parameter to any other process. Receiving processes can either pass the channel again or use it to send any data, as sending processes can read a value from it.

The original idea was to take advantage of the dual functions (`new/parfill`), using the latency on reading/writing to/from a channel as a sort of waiting/signaling subsystem. The main problem comes from the fact that Eden Channels are intended to communicate different processes (hence, threads) , but not to communicate threads inside a process. So, here is the trick: instead of sending the channels via ordinary process instantiation, as usual, we push/pop them into/from a persistent local queue (namely `Concurrent.Chan`) to the PE, i.e

```
----- ( Waiting Thread ) -----
new (\ chn (c::CInt) -> do
    queuep <- creadQueuePointer
    queue <- deRefStablePtr queuep
    writeChan queue chn
    return () 'demanding' Strategies.rnf c
----- ( Signaling Thread ) -----
    queuep <- creadQueuePointer
```

```
queue <- deRefStablePtr queuep
chn <- readChan queue
parfill chn (1::CInt) return()
```

To support these operations we must implement also some “atomic” operations on persistent variables saved on a external context, (FFI), namely the number of (implicit) threads waiting for enter the shared resource. (The code is well documented at `monitor.hs`, `monitor.c`)

7.2.3 MVar:

Of course, you can always use the well known concurrency abstraction from Concurrent Haskell, in the hope that underlying thread subsystem of Eden will “fit” that of Concurrent Haskell for synchronization issues (See 4.2.4) To do so, please, arrange the preprocessor directives as required.

7.3 Conclusion.

This has been the first Eden program where processes have been used to play a role other than symbolic processing, i.e, we had to deal with low-level system programming resources, such as “pipes”, “handles” and other POSIX services like “fork”...

As a consequence, we had to apply some patch (“tricks”) to solve the “paradigm mismatch” with Eden execution model, i.e, explicit synchronization on a pre-defined implicit threading subsystem. If implementing a local state relying on a “external context entity” (FFI) and further “monad bypassing” was rather an option at Concurrent Haskell, (you could always to export the monad to the final signature and overload it with pipe types...) in Eden it is mandatory, due to the disjoint memory local to each PE. This prevents you to pass a pipe handle from one process to other, since they can not be run on the same PE.