Declarative Scripting in Haskell

Tim Bauer and Martin Erwig

Oregon State University, Corvallis, Oregon, USA
{bauertim,erwig}@eecs.orst.edu

Abstract. We present a domain-specific language embedded within the Haskell programming language to build scripts in a declarative and type-safe manner. We can categorize script components into various orthogonal *dimensions*, or concerns, such as IO interaction, configuration, or error handling. In particular, we provide special support for two dimensions that are often neglected in scripting languages, namely creating deadlines for computations and tagging and tracing of computations. Arbitrary computations may be annotated with a textual tag explaining its purpose. Upon failure a detailed context for that error is automatically produced. The deadline combinator allows one to set a timeout on an operation. If it fails to complete within that amount of time, the computation is aborted. Moreover, this combinator works with the tag combinator so as to produce a contextual trace.

1 Introduction

Scripts are used for many different system maintenance tasks, such as program building, testing, system administration and various process management tasks. They are the "glue" that allows us to cobble programs into large systems.

Unfortunately, many consider scripting languages to be ad-hoc. They are "quick and dirty". Such scripts all tend to be imperative or procedural in their structure when in fact the problems they are addressing can be specified as well or better declaratively.

Another problem, shell scripts and `make`-file scripts rely on various executable programs such as `ls`, `awk` and `sort` to perform specialized computation; these programs are the cohesive force that holds the script together. This leads to two problems. First, such scripts are typically less efficient since multiple processes must be created for even the simplest tasks. More important, the semantics of those tools vary across different platforms making scripts using them brittle and non-portable. For example, `make` files frequently use extensions of some specific implementation such as `gnumake` or Microsoft's `nmake`. It would be far better to draw functionality from a stable system library in a language with consistent cross-platform semantics.

Debugging and error handling in scripting languages is not very well supported either. We are usually stuck with torrents of confusing output and cryptic error messages or the polar opposite: silent failure. Error messages in the subprograms like `awk` do not map very well to errors in the domain of the scripting language.

We also observe that traditional scripting languages lack a type system. Everything is just a string and we have no opportunity to detect errors before we start the script.

The above observations help us identify several orthogonal dimensions that scripting languages consist of: traceability, error handling and a type system. Viewing scripting tasks and programs in terms of these dimensions helps us with the design of the language, in particular, with assessing maintenance and scalability questions for scripts.

A less obvious dimension is configuration. For maintenance and portability scripts all tend to have constants declared at the top. The user needs to edit only the first few lines to make the script work correctly on their platform. In both shell scripts and make files we see the infamous structure given below.

```
... platform specific configuration

################################################
# DO NOT MODIFY ANYTHING BELOW THIS POINT
################################################

... stuff end user should not have to change
```

In this paper we present a domain-specific embedded language (DSEL) in Haskell for scripting. The language is defined on two layers. The bottom layer provides a broad set of generic script functions. On top of that, we define specialized functions to support particular domains. We then illustrate a use of this framework by using it to implement a grading language.

Our language framework allows for structured error handling, traceability and static type checking. Since it is embedded in Haskell, it draws from a host of stable libraries to use. Additionally, our framework contains a novel timeout or "deadline" facility allowing us to cleanly abort tasks that do not complete in a reasonable amount of time.

We support traceability with a special tagging combinator that permits us to label pieces of programs with descriptive annotations that would otherwise be lost to comment blocks. The annotations allow us to easily construct explanations when errors are encountered.

The outline for the rest of the paper is as follows. In section 2 we define the language framework. Here we discuss various decisions that have lead to our current design. Section 3 gives an instantiation of our framework in the domain of testing scripts. In section 4 we introduce our facility for traceability with annotating or "tagging" tasks. We also discuss error recovery in this section. In section 5 we introduce the useful deadline combinator allowing us to set timeouts on tasks that may enter infinite loops. Following this, in section 6, we extend the testing domain first introduced in section 3 to a more realistic example. Next, in section 7, we show how configuration is supported by our scripting framework. We present related work in section 8 and a conclusion in section 9.

2 Language Definition

Any scripting framework needs access to the outside world. As mentioned in the introduction, scripts frequently allow us to compose other programs through operating system features. In Haskell, this implies we need to be running in the context of the `IO` monad [16].

Additionally, we need to be able to store configuration much like platform-dependent constants that script maintainers keep at the top of their scripts. The `Reader` monad cleanly allows this capability by permitting one to maintain implicit well-typed read-only function parameters. Sub-actions can read and override the values, but they cannot change them for parents or siblings.

However, since we have identified two different monads our scripting environment must use, we must use monad transformers [3], specifically the `ReaderT` transformer.

These observations initially lead us to the following type.

```
type Task e a = ReaderT e IO a
```

We read this as: A `Task` with environment (configuration) type `e` and returning an `a` is a `ReaderT` monad (storing that `e`), with access to the `IO` monad, returning that same `a`.

Also recall we desire some clean, structured error handling. We have two options for error handling. First we could inject the venerable `ErrorT` monad transform into the mix, giving the following.

```
type Task e a = ReaderT e (ErrorT TaskException IO) a
```

(The type `TaskException` corresponds to our representation of an error.)

This approach is sorely tempting; each monad transformer corresponds to a discrete feature requirement. However, since we are working in the `IO` monad we have to deal with the `IO` monad's exception framework whether we like it or not [12, 17]. Hence, we would be stuck converting `IO` exceptions to monadic errors regardless. In fact, with Marlow's extensible exceptions [12] now in Haskell it is easier and more consistent to define our own specialized exception type and work within the `IO` exception hierarchy to handle errors.

```
data TaskException = TaskException Message Trace deriving Typeable

type Message = String
type Trace = [Tag]
type Tag = String

instance Exception TaskException
instance Show TaskException where
  ...
```

The exact type and function of `Message`, `Trace` and `Tag` are discussed later in section 4. Making the exception an instance of the `Show` and `Exception` type classes enable this data type to be a proper exception that we can uniquely

recognize and handle. This will prove quite valuable when we are propagating errors.

One final annoyance affects our monad's type, currently still a type synonym for `ReaderT e IO a`. The monad's `fail` method of a monad transformer delegates failure to the innermost monad's `fail` method, `IO` in this case. This results in an exception of an unknown type (`IOException`). However, we prefer calls to `fail` within our `Task` monad create and throw a `TaskException`. Hence, we need some way of "overriding" the `fail` method.

One way of doing this is to create a new data type that delegates all its work to the old type (except the `fail` method). Hence our `Task` type becomes the following.

```
newtype Task e a = T (ReaderT e IO a)
```

The monad's run function simply extracts the transformer and calls to the outermost (in the transformer stack) monad's run function, in this case `runReaderT`.

```
runTask :: e -> Task e a -> IO a
runTask e (T t) = runReaderT t e
```

We simply delegate most of the work to the inner reader monad and override `fail` for our own purposes.

```
instance Monad (Task e) where
  return   = T . return
  m >>= k  = T . ReaderT $ \r -> do { a <- runTask r m; runTask r (k a) }
  fail     = taskFail
```

The function `taskFail` is shown later.

3 A Testing Domain

To illustrate our new framework, we apply it to a concrete application, namely of program grading (or testing). Suppose we are grading student assignments in a class teaching Java. The students are given a skeleton source file and are supposed to write a simple factorial function. Our goal is to automatically grade this assignment with a test script.

Below is a sample solution in a file named `Asgn1.java` by student Jim.

```
class Asgn1{
  static int fact(int n) {
    return ((n == 0) ? (1) : (n * fact(n - 1)));
  }

  public static void main(String[] args) {
    System.out.println(fact(Integer.parseInt(args[0])));
  }
}
```

The student's file includes a simple `main` method that parses an input argument from the command line, applies their `fact` method to it and writes the result to standard output. We assume this file is stored in the directory `handin/Jim`. (Other students' submissions would be in sibling directories such as `handin/Sara`.)

Our grading script must compile the student's source file, execute it against various inputs and match the output with some expected output. In the following we demonstrate how we could write a testing script with our framework.

First, we instantiate our `Task` monad to this domain, by specializing it as a `Test` monad. For now, we do not need any environment so we can ignore that parameter with `()`.

```
type Test a = Task () a
```

We also introduce some useful type abbreviations and define the type of a `Result`. A test result can be `Correct` indicating the test succeeded, it can be `Incorrect` indicating it is wrong, or one of a several other specialized values indicating another sort of error.

```
type Student = String
type Input   = String
type Output  = String

data Result = Correct
            | Incorrect Output
            | TimedOut
            | CompileFailure
```

The function `simple` constructs a very simple test case by composing a compilation task `simpleCompile` and a task `run`, which checks various inputs and compares them to the expected value.

```
simple :: Student -> Test Result
simple s = do { simpleCompile s; run s ("4","24") }
```

The `simpleCompile` function takes a student name and compiles that student's file using the `exec` combinator, which actually forks a process (`javac` in this case). The `exec` task succeeds only if the underlying process returns a zero exit code; `javac` will do this as long as there are no compilation errors.

```
simpleCompile :: Student -> Test ()
simpleCompile s = javac s "Asgn1.java"

javac :: Student -> FilePath -> Task e ()
javac s f = exec ["javac", "handin/"++s++"/"++f]
```

The `run` function referred to above runs a program on a given input and compares the output (with whitespace trimmed by `trimWS`) against some expected output with a `match` function.

```
run :: Student -> (Input,Output) -> Test Result
run s (inp,expct) = do out <- java s "Asgn1" inp
                       return (match expct (trimWS out))

match :: Output -> Output -> Result
match expct o | o == expct = Correct
              | otherwise  = Incorrect o
```

Finally, the `java` task starts a `java` process and returns the standard output stream as `Output`.

```
java :: Student -> FilePath -> Input -> Test Output
java s f i = exec ["java", "-cp", "handin/"++s, f, i]
```

The alert reader might note that the `exec` function is returning `()` when used in `javac` and `Output` here. This is possible since the function is overloaded in its return type [21]. That is, the type is

```
exec :: ExecResult r => [String] -> Task e r
```

If output is needed from the process such as in the `java` task above, such an implementation is selected by the type system. If output is not needed as in this case, a version that returns `()` is used.

Finally, to finish our example, we include a simple function `runTest` to run the monadic computation and return the result. Recall `runTask` requires an initial value for the environment, but since we do not need one yet, we can ignore it by passing the `()` value.

```
runTest :: Test a -> IO a
runTest = runTask ()
```

To test Jim's assignment, we call this function. The test passes successfully as indicated by the return value.

```
*Main> runTest $ simple "Jim"
Correct
```

We will extend this example in later sections as we introduce more features of our framework such as error handling and deadline support.

The functions that we have defined in this section build on the underlying scripting framework and encode functionality that is pertinent to the particular domain, testing.

We can see here that embedding the scripting language as a DSEL in Haskell is advantageous because we can use all the convenient language features of Haskell to define functions quickly and concisely. We can also easily spot opportunities for generalizations.

4 Task Annotations and Errors

One of the key features of our scripting framework is our error handling support. As we bemoaned before, scripting environments give us very little (often no)

information or way too much information. For example, `bash`'s `-v` option will echo every line executed. But this is akin to having an imperative program echo every statement executed, or having a functional program tell us every graph reduction that occurs. You see everything, but cannot determine why; there is no explanation for the actions.

This is insufficient. Instead, it seems better to have the programmer decide what is of interest and annotate what is going on from the highest to the lowest level. For this we introduce a `tag` combinator to annotate tasks.

```
tag :: Task e a -> Tag -> Task e a
```

We illustrate this combinator with two examples given in the next sections.

4.1 A Startup Script

Suppose we want a script to start a server and client program. The script must update some source code for both programs to the newest version by copying it via `scp`, it must compile the code with `gcc` and then start the client and server.

Using the `tag` combinator and Haskell's infix function application, this might look as given below.

```
start = do
  update "server" `tag` "getting and compiling server"
  update "client" `tag` "getting and compiling client"
  server          `tag` "starting server"
  client          `tag` "starting client"

update f = do
  scp ("user@somehost:"++f++".c") `tag` ("getting file "++f++".c")
  gcc f                           `tag` ("compiling "++f)
```

The tasks `scp`, `gcc`, `server` and `client` are self explanatory calls to `exec` similar to `java` and `javac` given earlier.

Now suppose we move this script to a machine that does not have `gcc` installed. The error output is useful to all range of experts, from system administrators (non-programmer) to the script maintainers.

```
*Main> runTask (start `tag` "top-level") ()
*** Exception: TaskException: gcc: runInteractiveProcess: does not exist
(No such file or directory)
  from exec [gcc client.c -oclient]
  from compiling client
  from getting and compiling client
  from top-level
```

Note that the `tag` combinator provides a second powerful purpose as shown in the client-server script above: It effectively combines the function of an end-of-line comment with an annotation that can be used at runtime to explain to an end user what is going on. We kill two birds with one stone: documenting our code and reporting context at the appropriate level in the event of an error.

Script errors, `IO` exceptions or calls to the `Task` monad's `fail` method, are implemented as a special exception. Recall in the introduction we showed `fail` as an opaque call to `taskFail`. We now define that function below.

```
taskFail :: Message -> Task e a
taskFail m = liftIO . throwIO $ TaskException m []
```

It takes an error message and converts it to an exception in the `IO` monad which it throws (with an empty context). That exception gets threaded upwards. Any call to `tag` will catch the `TaskException`, append the new program annotation to the trace and then propagate the new exception on upwards by rethrowing it.

Contrast our scripting framework's tagging approach to languages such as Python or Java that emit stack traces when an unhandled error is encountered. While these traces are automatically generated (no explicit tagging is required), they have other disadvantages. For one, they are more verbose than we desire, many of the frames may not be helpful in explaining what a program is doing. Similarly, the function names are compile-time constants; our tags can be generated dynamically and include runtime information.

4.2 Some Comic Relief

Another example of a task that makes use of tags is given here. Our goal is a light-hearted shell-script that fetches a webpage containing a cartoon using `wget`. It scans the page using a regular expression for the image URL of the most recent comic, fetches that image URL and loads the it in am image viewer.

All sorts of errors can occur here. For instance, the network could be down, the target URL might not exist anymore, the script might be run on a machine without `wget`, or the format of the page could change.

In our library we can implement a function `readUrl` as follows.

```
readUrl :: String -> Task e String
readUrl url = exec ["wget", "-O-", url]
```

As any well-behaved Unix program does, `wget` sends error messages to `stderr` and produces a non-zero exit code in such cases; the combinator `exec` will see the non-zero exit code and propagate that error messages as an exception in in the domain of our scripting language.

The next piece required is regular expression matching. In our library this is implemented as a function that uses `Text.Regex` from the Haskell hierarchical libraries [11].

```
matchRegex :: Pattern -> String -> Task e String

type Pattern = String
```

The first argument to `matchRegexTask` is a regular expression pattern string, the second is the string to match against. The computation results in the first matching instance of the given pattern. If no pattern is found, a sensible error

message is produced. Similarly, if a malformed pattern is given, an error message indicating that is produced.

These pieces combine nicely into the following script.

```
xkcd :: Task e ()
xkcd = do
  page    <- readUrl "www.xkcd.com"     `tag` "reading XKCD webpage text"
  imgUrl  <- matchRegex urlPattern page `tag` "matching image URL"
  imgFile <- getUrl imgUrl              `tag` "fetching image"
  exec [imgViewer, imgFile]             `tag` "loading image viewer"
  where imgViewer  = "firefox"
        urlPattern = "http://imgs\\.xkcd\\.com/comics/[a-zA-Z0-9_]+\\.png"
```

In this code `readUrl` will fetch the current XKCD comic's webpage (using `wget`), `matchRegex` will scan the HTML contents for the direct URL to the latest comic, `getUrl` will fetch that file (again using `wget`), and the final task expression will launch an image viewer (`firefox`) to view the image.

Any of the above `Task`s could fail. In each case, we use the `tag` combinator to provide context as well as document what that sub-task is doing. Sample output from the above script is given for the case where fetching the webpage fails.

```
*GetComic> taskRun () xkcd
*** Exception: TaskException: exec [wget -q -O- www.xkcd.com] exited 1:
  from exec [wget -q -O- www.xkcd.com]
  from fetching XKCD webpage text
```

The output is relatively concise and clear. The low-level message indicates that the `readUrl` task failed since `wget` exited with a non-zero code. The high-level message tells us we were doing the above task because we were trying to fetch the page to find the image URL.

Now consider a similar script in `bash`.

```
#!/bin/bash
page=`wget -q -O- "www.xkcd.com"`
imgurl=`expr match "$page" \
        '.*\(http://imgs\.xkcd\.com/comics/[a-zA-Z0-9_]*\.png\).*'`
wget $imgurl -Oimg.png
firefox img.png
```

If the above error is encountered (failure to find page), the shell ignores the failure code, evaluates the `page` variable to the empty string and optimistically plunders onward. That empty variable causes the `expr` process to fail. What happens now? The `imgurl` variable becomes the empty string and the next `wget` command fails. By the time we notice the failed behavior, the script is nowhere near its point of failure.

One can tell `bash` to stop on the first error (non-zero exit code) and trace commands executed by setting the options `-evx`. The output, given below, indicates the correct point of failure by showing command evaluations up to the point of failure.

```
page=‘wget -q -O- "www.xkcd.com"‘
wget -q -O- "www.xkcd.com"
++ wget -q -O- www.xkcd.com
+ page=
```

We feel our output is more useful since it has the capacity to include higher level error messages as shown earlier.

4.3 Error Recovery

We have introduced an effective error reporting and script documenting mechanism. Here we introduce ways our framework offers to recover from errors rather than just aborting the script (the default behavior). To achieve this we introduce several combinators.

The `handleFailure` combinator takes a task computation and an error handling function. If the task encounters some error, the handler is run.

```
handleFailure :: Task e a -> Handler e a -> Task e a

type Handler e a = (Message,Trace) -> Task e a
```

This combinator is similar to `handle` or `catch` in Haskell's `IO` exception handling library `Control.Exception`. However, unlike its counterpart, handlers in our language receive contextual information they can potentially use.

Recall our grading example started in section 3. Suppose a student Mike misnames their submission file (we expect it to be named `Asgn1.java`). In that case an error occurs, the `javac` process returns a non-zero exit code, and the `exec` task converts this into a call to the `fail` method of the `Task` monad. In short, the error is unhandled and kills the script (albeit with useful information).

```
*Main> runTest $ simple "Mike"
*** Exception: TaskException: non-zero (2) exit code
  from exec [javac handin/Mike/Asgn1.java]
```

It would be more desirable to consider a misnamed file a test failure and return a value `CompileFailure` rather than crashing the entire script. Below we show a function `safer` which does exactly this.

```
safer s = simple s ‘handleFailure‘ (\_ -> return CompileFailure)
```

This function will now properly handle test cases that fail.

```
*Main> runTest $ safer "Mike"
CompileFailure
```

We use this functionality extensively later when we extend our test language example.

5 Deadlines

Recall we defined process management as one of the requirements that scripting languages must address. Be it `make` or a shell script, process management is vital. For example, in the grading script we spawned `javac` and `java` to compile and test a student's program.

Certainly, in the grading domain a veritable possibility is that a student program will enter an infinite loop. In such cases our grading scripts will never halt. But since all student programs should finish within a few seconds, we can stop those that do not, considering them instances of failure. To achieve this we introduce a ternary operator composed of two functions `timeoutIn` and `onTimeout` whose signatures are given below.

```
type Seconds = Float

timeoutIn :: Seconds -> Task e a -> TimedTask e a
onTimeout :: TimedTask e a -> Task e a -> Task e a
```

A `TimedTask` is an opaque type. The only way to create one is via `timeoutIn`. The only way to get a `Task` back (to run or modify) is to use the handler `onTimeout`. This allows us to statically ensure that a handler can only be attached to a timed task and that any timed task has a timeout handler. Uses always looks of the form given below.

```
longTask `timeoutIn` seconds `onTimeout` timeoutValue
```

We are effectively creating a ternary operator from Haskell's infix syntax with this construct.

A realistic use of this can be shown in our testing domain. We define a `limit` function which limits the time a task can run before it must be considered a timeout failure, which is represented by the constructor `TimedOut`.

```
limit :: Test Result -> Seconds -> Test Result
limit t s = t `timeoutIn` s `onTimeout` (return TimedOut)
```

We illustrate use of this combinator below by considering a student Tom whose factorial function loops infinitely on certain inputs. The code below never terminates.

```
*Main> runTest $ safer "Tom"
```

Clearly, this is a problem if we expect to be able to test multiple students' programs in one script.

The above expression can be fixed easily using the `limit` operator we defined above. Now the simple code below terminates as we desire in after 4 seconds.

```
*Main> runTest $ safer "Tom" `limit` 4
TimedOut
```

Since we are annotating (tagging) our programs with the `tag` combinator as described earlier, we can not only handle deadlines, but even report what we were doing when we timed out. To do this we introduce an additional `withDeadline` operator.

```
withDeadline :: Task e a -> Seconds -> Task e a
```

This combinator is similar to `onTimeout` except no handler is required. Instead timeout errors are just converted into regular `TaskException`s and passed on upwards. However, the part of the `Trace` that timed out is annotated.

To illustrate, this effect we show a contrived example whereby we nest several tasks. Figure 1 shows the tree corresponding to the script given below.

```
a = do {b; c} `tag` "a"
c = do {d; i} `tag` "c"
d = do {e; f} `withDeadline` 2 `tag` "d"
f = do {g; h} `tag` "f"
g = infiniteLoop `tag` "g"
```

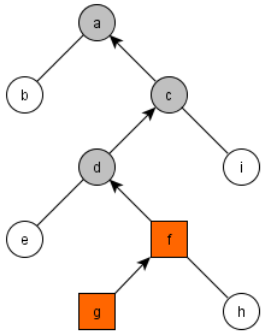

Fig. 1. A tree of tasks

When this script is run, execution will proceed as follows: task `a` will call task `c`, which calls `d`. Task `d` installs the `withDeadline` timeout handler and eventually calls `f`, which calls `g`. Task `g` loops infinitely. However, the deadline handler in `d` stops this after two seconds. The result output is shown below.

```
*Main> taskRun () a
*** Exception: TaskException: deadline expired
  from g <- timed out
  from f <- timed out
  from d
  from c
  from a
```

Note how the tags below the `withDeadline` handler in task `d` (`g` and `f`) are annotated indicating that they are part of the task that timed out. Now we can tell that the `a` task failed because one of its descendents (`d` in this case) timed out. Moreover, we also get to see what task `d` was doing when it timed out (calling into `f` and `g`).

6 Extending the Testing Subdomain

With the infrastructure in place we can extend our testing example reusing much of the code already introduced. Our end goal will be to introduce a script that grades several student programs while gracefully and concisely handling errors such as misnamed files and programs that enter infinite loops.

First we require some definitions for test cases.

```
type Points = Int
type Label = String

type Case = (Test Result, Bool, Label, Points)
```

A test `Case` is a four tuple consisting of a `Test Result` computation as defined earlier in section 3, a flag to indicate if that case is critical, a label and point value. The meaning of these attributes is described later. A test script is then just a list of these test cases.

```
type Script = [Case]
```

The meaning of a test script is defined by the function `evalScript` that evaluates the script case by case until it either reaches the end or encounters a test case that fails, which happens whenever the `Result` is not the `Correct` constructor.

```
type CaseResult = (Result,Label,Points)

evalScript :: Script -> Test [CaseResult]
```

The value of `Points` returned will be the points accumulated by running that case. The student's grade is a sum of these point values. Since the `evalScript` function must actually run the monadic computations, it must work in context of the `Test` monad. The implementation of that function is long but straightforward and thus not shown here.

6.1 A Syntax for Test Cases

We can distinguish between quite a few different kinds of test-case attributes. A test case can be critical. If a critical test cases fails, the whole test script fails. A compilation test case would be an example of a critical case; there is no point trying to run the student's program if it fails. A test case can be timed or un-timed. A timed test has a deadline handler applied to it. Certainly, running a student's program should be time limited, but a compilation test does not need to be timed, we trust the compiler. Finally, a test case can have a text label identifying it as well as an optional point value.

All these variations and options can be summarized in the following informal syntax for a miniature DSL.

`[critical] [timed] test [labeled` *str*`] [worth` *pts*`] with` *expr*

The non-terminal *str* evaluates to type `Label`, *pts* is a Haskell expression of type `Points` and *expr* evaluates to a `Test Result`. Square brackets indicate optional constructs. All test case attributes have default values. By default cases are not critical, not timed, labeled with the empty string and worth zero points.

Keywords (indicated in typewriter font) correspond to functions that transform a `Case`'s parameters. Hence, complex test descriptions may be built via function composition, the `(.)` operator in Haskell. The expression below composes a function that creates a critical, timed test worth 10 points.

```
critical . timed . test . worth 10
```

We impose our syntax with Haskell's type system. For example, the `critical` modifier precedes the `test` keyword, and the `test` keyword must precede any `labeled` clause. Finally, the `with` clause should be given last. We can achieve this by introducing a new type.

```
type PreCase = (Test Result,Label,Points)
```

A `PreCase` is just `Case` but lacking information about whether that case is critical or not.

The `with` function given below is the right-most function in our test case's syntax. Recall its argument *expr* is a `Test Result` computation.

```
with :: Test Result -> PreCase
with tr = (tr,"",0)
```

It populates a `PreCase` with default values.

The `labeled` function takes a label and a `PreCase` such as that produced by `with` and overrides the old value of the `label`.

```
labeled :: Label -> PreCase -> PreCase
labeled l (tr,_,p) = (tr,l,p)
```

The optional clause `worth` is almost identical to `labeled`.

The only way to turn a `PreCase` into a `Case` is via the `test` function "keyword". This ensures this keyword appears. The implementation simply sets the critical flag to its default value of `False`.

```
test :: PreCase -> Case
test (tr,l,p) = (tr,False,l,p)
```

The `critical` function corresponds to that modifier and sets that value to `True` in the `Case`.

```
critical :: Case -> Case
critical (tr,c,l,p) = (tr,True,l,p)
```

The `timed` modifier is similar. Note, we hardwire the timeout at 3 seconds. We will generalize this later.

```
timed :: Case -> Case
timed (tr,c,l,p) = (tr `limit` 3,c,l,p)
```

With this approach, we can see that Haskell's type system ensures that we obey our desired syntax. For example, the case below is illegal.

```
critical . with
```

The `test` keyword is missing. This is a type error since `critical` requires a `Case` and `with` instead produces a `PreCase`.

As given so far our test-case implementation permits two deviations from the specified syntax: the order of `critical` and `timed` keywords may be interchanged and modifiers may be repeated. Both problems are easily solved with additional intermediate types (like `PreCase`) and more elaborate schemes, but are beyond the scope of this simple example. Other methods of embedding syntax are described or used in other work such as [15, 13, 14].

6.2 Building Test Cases

An example script shown below is built out of several test cases and allows us to fully test a student's program.

```
script :: Student -> Script
```

The script starts with a critical compilation `Case` which really illustrates the spirit of our small testing language. Compilation is a critical case; if it fails, we do not try and run the student's code. Following this are several cases that test various inputs on the student's program. Each of these is non-critical, just because one fails does not mean the others will necessarily fail.

```
script :: Student -> [Case]
script st = [ critical . test . labeled "compile" . worth 10 . with $ compile st,
              (0,  1) `testRunWorth` 40,
              (4, 24) `testRunWorth` 25,
              (6,720) `testRunWorth` 25]
  where testRunWorth :: (Int,Int) -> Points -> Case
        testRunWorth (i,o) w = timed . test . labeled (show i) . worth w
                                     . with $ run st (show i, show o)
```

The `compile` function from the compilation case above is similar to the one given earlier except that it now produces a `Result` value.

```
compile :: Student -> Test Result
compile s = c `onFailure` (return CompileFailure)
  where c = javac s "Asgn1.java" >> return Correct
```

Lastly, the `run` function was given earlier in the paper. Recall that it just executes the student's program passing input on the command line and compares the output to some expected value.

A benefit to embedding our scripting language in Haskell is now visible. Higher-order functions make it so we do not have to replicate syntax for each student if we want to test them all. Since `script` is parameterized by a `Student` we are effectively generating a custom test script for each student.

To run the `script` for all students one could use the following function.

```
testAll :: [Student] -> Test ()
testAll sts = forM_ sts $ \st ->
  evalScript (cases st) >>= liftIO . putStrLn . ppCaseResults st
```

The `forM_` function is Haskell's built-in "for each". It takes a list of values and applies to each a monadic function. For us here that is simply passing a `Student` to the `evalScript` line for each student. The pretty printer `ppCaseResults` is not given here, but is very simple.

Sample output from running `testAll` is given below.

```
*Main> runTest $ testAll ["Jim","Mike","Sara","Tom"]
Jim's fact -> 100
  compile  ->       passed     +10
  0        ->       passed     +40
  4        ->       passed     +25
  6        ->       passed     +25

Mike's fact ->   0
  compile  -> compilation failure

Sara's fact ->  35
  compile  ->       passed     +10
  0        ->       failed
  4        ->       passed     +25
  6        ->       failed

Tom's fact ->  50
  compile  ->       passed     +10
  0        ->       passed     +40
  4        ->    timed out
  6        ->    timed out
```

One can easily envision more elaborate output or follow-up actions, perhaps generating reports, gradesheet webpages or updating spreadsheets. Within our scripting framework such extensions are easy to define.

7 Environments

Recall earlier that we discussed the need for configuration in scripting. Many scripts start with large blocks of platform-specific configuration. The script user need only modify those few constants at the top to use it. To support this scripting dimension our `Task` monad composes a `ReaderT` transformer. The `e` type parameter of `Task` is exactly our configuration's type.

We show a simple example of how such a configuration can be used. Recall the `timed` function in section 6.1. It hardcoded the timeout of 3 seconds in its call to `limit`.

```
timed :: Case -> Case
timed (tr,c,l,p) = (tr `limit` 3,c,l,p)
```

Now suppose we want to make this timeout value a configurable parameter. Only three changes are required. We must indicate the new configuration type by changing the type of `Test`'s environment.

```
type Test a = Task Seconds a
```

Second we must change `timed` to use that value. We lookup the value of the configuration via the `ask` method of `MonadReader` (of which `Task` is an instance).

```
timed :: Case -> Case
timed (tr,c,l,p) = ((tr `limit`) =<< ask,c,l,p)
```

(The `=<<` operator is just `>>=` with the arguments interchanged.)

The final change is to specify an initial value at the top level. We do this by changing `runTest`.

```
runTest = runTask 10
```

This effectively sets the timeout to be 10 seconds.

It is worth noting that the configuration type may be overridden. This allows us to switch into combinators that have different environment requirements. Our library offers this with the function `reconfigWith`, which maps an old environment `e` into a new environment `f` for a given test case.

```
reconfigWith :: Task f a -> (e -> f) -> Task e a
```

We now illustrate this function by extending our previous example. Suppose we want to upload the grading results for each student to a server. To support this imagine another tasked-based API to support secure command-line operations, namely those that require a password.

```
type Secure a = Task Password a
type Password = String
```

This small API works in our `Task` monad by assuming the password is stored as the environment. This way, it does not have to interactively prompt the user for their password when it is needed for multiple operations. For example, there might be an `scp` (secure copy) combinator in this library as well as the venerable `ssh`.

```
scp, ssh :: [FilePath] -> Secure ()
```

Somewhere in the bowels of these functions that password is fetched via `ask` and passed to the standard input stream of the secure command when the password is needed.

We make use of this new secure task environment by introducing a variant of `testAll` which we call `gradeAll`. This function has an augmented task environment: it needs the timeout for the grading script, and it also needs the password to handoff to the network library's environment.

```
gradeAll :: [Student] -> Task (Seconds,Password) ()
gradeAll sts = forM_ sts $ \st -> do
   crs <- evalScript (script st) `reconfigWith` fst
   uploadGrades st crs           `reconfigWith` snd
```

The `evalScript` function used above is unchanged from before.

The `uploadGrades` function pretty prints the script result, copies it to the server (via `scp`) and sets the permissions on the file to be readable by students (via `ssh`).

```
uploadGrades :: Student -> [CaseResult] -> Secure ()
uploadGrades st crs = do
  let report = "report_"++st++".txt"
  liftIO $ writeFile report $ ppCaseResults st crs
  scp report "me@myserver:public_html/grades"
  ssh ["me@myserver", "chmod", "o+r","public_html/grades/"++report]
```

8 Related Work

The benefits of embedding DSLs in Haskell are numerous and have been described, for example, by Hudak [5, 6] as well as Leijen and Meijer [10]. Haskell allows for more readable embedded DSL programs with its infix function application, and monads provide a powerful abstraction of computations. Moreover, we get the full power of Haskell's type system and predefined libraries. This makes it appealing for scripting tasks usually relegated to ad-hoc shell scripts and make files.

We illustrated a simple subdomain for testing programs. Within Haskell considerable work has already been done in embedded testing languages. However, much of the focus is on testing only Haskell programs within Haskell. For example, QuickCheck [4] is a Haskell tool that allows a user to write small invariants that should hold as Haskell functions. QuickCheck then uses the types of the functions being tested to automatically generate various inputs (generated inputs can also be constrained). With this approach the "startup" cost of writing tests is extremely low, one can find a *witness* input that breaks expected invariants with almost no work.

Smallcheck [18] extends the idea behind QuickCheck by generating all the possible test inputs up to a certain depth in the type tree instead of just random ones. QuickCheck and SmallCheck are restricted to testing Haskell programs only. In contrast, our language focuses more on helping one build complicated testing hierarchies (or general scripting language tasks). Moreover, since our language is embedded in Haskell, both QuickCheck and SmallCheck could be used very easily to great effect.

A nice parallel between Unix shell operators and monads in Haskell was drawn in an essay [8] by Oleg Kiselyov. The bind function `>>=` corresponds roughly to the pipe operator in shells, the `return` (unit) function corresponds to the `cat` (catenation) program. This idea inspired a project called `Haskal` by Jansborg and Despotoski [7]. The project consists of a small shell that supports simplistic command pipelines as well as some other features.

Stewart's `h4sh` [20] shell implements many Haskell prelude functions usable from a command-line shell. For example a program's output is treated as list of

strings, and one may build elaborate pipelines using Haskell code with familiar functions such as `filter` and `foldl`.

Hashell is a shell language written in Haskell to allow Haskell expressions to be intermixed with a command pipeline [2]. One escapes the Haskell expressions from the rest of the pipeline. Sadly, there is almost no documentation on this project and only several examples.

Scheme shell [19] introduced by Shivers implements a shell in the language Scheme. The shell allows for low-level access to C system calls as well as permitting higher-level I/O redirection and pipelines. The shell makes use of Scheme's powerful macros in its implementation. However, there are several differences. Scheme is not statically typed, nor is the language purely functional. I/O and variable mutation are permitted in any context, thus scripts in that shell are less declarative.

The Parsec parser combinator library [9] includes an annotation combinator `<?>` very similar in purpose to our `tag` combinator, that is, supporting contextual error messages.

9 Conclusion

We have presented a core DSL for scripting that can be easily extended into DSLs for various application domains in scripting. We have shown how it fulfills many tasks previously relegated to imperative shell scripting languages in a declarative manner.

We introduced a novel deadline abstraction combinator that allows us to cut off run-away computations. While this can be emulated in shell scripts, it is neither simple nor portable. Additionally, our scripting framework supports high-level annotation (tagging) of arbitrary tasks. What would be simple program comments can be used to offer an explanation from the highest to lowest levels of abstraction if an error is encountered. Our scripting framework also allows for a top-down environment for configuration information that scripts (shell-scripts and makefiles) commonly have.

Since our language is embedded in Haskell, our scripts are statically typed. Additionally, the embedding gives scripts access to an incredible amount of freely available (and well tested) code with *stable* semantics across multiple platforms. Contrast this with the numerous versions of helper programs that shell scripts rely on and their inconsistent semantics. Moreover, we illustrated the seamless use of these libraries showing several combinators that use the Haskell hierarchical libraries that automatically come with GHC such as regular expressions and process management libraries.

Our implementation of this project is freely available online [1].

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments.

References

1. HDDS - Haskell Domain-Specific Scripting Framework. `http://eecs.oregonstate.edu/~bauertim/hdds/`.
2. L. Araujo. Hashell. `http://www.haskell.org/hashell/`, 2005.
3. R. S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall International, London, UK, 1998.
4. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
5. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.
6. P. Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
7. M. Jansborg and A. Despotoski. Haskal — the haskell shell. `http://www.cs.chalmers.se/~dave/Courses/Topics/SavedProjects/`, 2003.
8. O. Kiselyov. Monadic i/o and unix shell programming. `http://okmij.org/ftp/Computation/monadic-shell.html`, 2003.
9. D. Leijen. Parsect, a fast combinator parser. `http://www.cs.uu.nl/~daan/parsec.html`, 2001.
10. D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, Oct. 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).
11. H. H. Libraries. `http://haskell.org/ghc/docs/latest/html/libraries/`.
12. S. Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 96–106, New York, NY, USA, 2006. ACM.
13. E. Meijer and D. van Velzen. Haskell server pages: Functional programming and the battle for the middle tier. In *In Proceedings Haskell Workshop*, 2000.
14. C. Okasaki. Techniques for embedding postfix languages in haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 105–113, New York, NY, USA, 2002. ACM.
15. J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 91–105, London, UK, 1998. Springer-Verlag.
16. S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2001.
17. S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *In SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36. ACM Press, 1999.
18. C. Runciman. Smallcheck. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/smallcheck`, 2007.
19. O. Shivers. A scheme shell. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1994.
20. D. Stewart. h4sh. `http://www.cse.unsw.edu.au/~dons/h4sh.html`, 2005.
21. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76, New York, NY, USA, 1989. ACM.