Experimentation Project: Safer Shell Scripts Using Dependent Types

Cas van der Rest

November 2018

1 Introduction

Little safety is provided when executing third party shell scripts. Usually there is no way to know anything about the effects of a script on a system without thorough inspection of its contents, a task we would preferably avoid. Furthermore, a script is often executed with more *capabilities* than it needs. In the context of a shell, a script usually receives whatever authority the user that executes it has on the system, a concept known as *ambient authority.* Depending on the role of the user this in itself can be problematic, were it not for the fact that it is not uncommon for a user to execute scripts with root privileges in case the script needs to modify something that is outside the user's authority; especially installation scripts suffer from this problem.

Execution of a script from an outside source would proceed with much more confidence if it would provide some kind of metadata describing its effects, in a format that is easy for a user to inspect. Of course, this only works if we know that a script will not act outside what is described in its metadata.

This project is an exploration of how these issues can be mitigated by embedding shell scripts into Idris [2], using its dependent type system to model a script's behaviour and statically prevent mistakes or undesirable behaviour. Various techniques and their application to this problem are discussed.

2 Related Work

This project is largely motivated by *Shill* [1], a scripting language developed at Harvard University. Shill is based around the *principle of least privilege* [3] (a script should have no more authority than it strictly needs), and takes a sandbox-based approach to enforcing this principle.

Every Shill script comes with a contract, describing the *capabilities* of script (i.e. the resources it requires to run). The sandbox will only allow a script access to resources that are part of it's capabilities. Similarly, native shell commands that are called from a Shill script are also executed in the sandbox, and thus are restricted in the same way.

An example contract of a script taking one input parameter called $input\_file$ could be:

$$
\begin{array}{l}
\textit{provide} : \\
\quad \{\, \textit{input\_file} : \textit{is\_file} \wedge \textit{writable} \,\} \rightarrow \textit{Void}
\end{array}
$$

proclaiming that the input parameter should refer to an existing file, and that the script will need write permissions on that file. Shill contracts consist of a precondition and a return type. An example of a script with the above contract could be:

$$
\begin{array}{l}
\textbf{if } \textit{is\_file}\ (\textit{input\_file}) \wedge \textit{has\_ext}\ (\textit{input\_file}, \texttt{"jpg"}) \textbf{ then} \\
\quad \textit{append}\ (\texttt{"Hello, World!"}, \textit{path}\ (\textit{input\_file}));
\end{array}
$$

Although Shill's API provides the necessary tools to specify fine-grained authority for scripts, all enforcement of contracts happens dynamically. This comes with an obvious drawback: how do we deal with scripts that fail dynamically halfway through their execution? Preferably we would reject such scripts statically, preventing their execution at all. Also notice that $\textit{is\_file}\ (\textit{input\_file})$ appears both in the contract and the script itself.

3 Project Scope

Bash is a *very* elaborate shell, and to try to capture all its nuances in this project is clearly not a reasonable objective. At the very least, we would like to cover some very basic scenarios where scripts try to access files or directories. An example of such a script would be:

```
ls /home/cas
cat /etc/shadow
```

The corresponding Shill contract would look something like the following:

$$
\begin{array}{l}
\textit{provide} : \\
\quad \{\, \texttt{"/home/cas"} : \textit{is\_dir} \wedge \textit{readable} \\
\quad ,\ \texttt{"/etc/shadow"} : \textit{is\_file} \wedge \textit{readable} \\
\quad \} \rightarrow \textit{Void}
\end{array}
$$

We can identify several properties of files and directories we would like to be able to specify and assert by means of a precondition. Most notably whether a resource is an existing file or directory and if a user has certain authority over that resource.

Merely a precondition is obviously not sufficient to specify more complex behaviour. In particular, dependencies between different parts of the script can be hard to capture. Consider the following snippet:

```
touch file.txt
cat file.txt
```

There is a clear dependency here that is not easily captured in a precondition: `cat file.txt` needs the previous command to execute. Requiring that `file.txt` exists beforehand would be too strong. For simplicity sake, we will initially assume that no such dependencies exist within a script.

3.1 Command Line Options

The behaviour of a command (and by extension the required parameters and return type) often depends on the various flags and options that were specified. Invoking the `man` command for any of the more common commands reveals a vast array of possibilities. To circumvent the problems this implies for a formalization of a command's behaviour, we assume a simplified model in which any single command has to have a fixed set of parameters and return type.

4 Codebase Overview

Although the codebase is relatively small, this section contains a concise overview of the files contained in the repository, for convenience of those who want to lookup and play around with the code listed in this report. We limit ourselves to the *V3* directory containing the most complete version.

- `AtomicProofs.idr`: Contains functions that help generate proofs that a file exists and that it has certain properties.
- `CmdIO.idr`: Provides a backend that allows for execution of scripts using the *IO* monad.
- `Environment.idr`: Contains all datatypes relating to the filesystem, and their corresponding implementations of *DecEq* (decidable equality).

- `Eval.idr`: Brings all the components together into a few simple example scripts.
- `Free.idr`: Implementation of the *Free* datatype for Idris.
- `Parsers.idr`: Parsers used by the *IO* backend.
- `Syntax.idr`: Syntax definition of the scripting language and precondition calculation.

5 Datatypes and Proofs

5.1 File System

If we aim to reason about the effect of a script on a filesystem, it is convenient to have some kind of abstract representation. This allows both the assembly and proving of a script's precondition to happen independent of the implementation details of the underlying filesystem.

5.1.1 The File System Tree

The chosen representation is a rose tree with an additional constructor for leafs, in order to be able to distinguish between files (leafs of the tree) and empty directories (nodes with no children). Both nodes and leafs contain the name and metadata for the directory or file at that location. The contents of a file are not included; we cannot reasonably expect a tree representing an entire filesystem to fit into main memory in that case.

This results in the following datatype definition:

$$\begin{aligned}\textbf{data}\ \mathit{FSTree} &= \mathit{FSNode}\ \mathit{FileInfo}\ (\mathit{List}\ \mathit{FSTree})\\ &\mid\ \mathit{FSLeaf}\ \mathit{FileInfo}\end{aligned}$$

It should be obvious that leafs are only meant to contain files, and nodes are supposed to contain directories. This is not enforced by the datatype itself, but in practice the trees used will probably represent some existing filesystem, so it seems reasonable to assume that no such mismatch will occur.

5.1.2 File Metadata

A small amount of metadata is recorded for both directories and files. We restrict ourselves to properties that actually influence a user's authority. Properties that have little to do with authority over a file, such as the date it was last modified or the filesize, are excluded. We use the following datatype:

data *FileMD* : *Type* **where**
  MkFileMD : $(t : FType) \to (p : FPermission) \to (u : User) \to FileMD$

FPermission simply mirrors the permission model that is commonly found in UNIX-like systems: 9 bits in total, with 3 groups of 3 bits, one for the file owner's permission, one for the file owner's group's permission and one for the permission of others. The three bits per group mark read, write and execute permission respectively.

The possible types are limited to files ($F\_$) and directories ($D\_$). To keep things maneageble, symlinks are ignored as they complicate the *FSTree* datatype quite a bit.

5.2 Predicates

The programmer may use standard predicate logic to express a commands behaviour. As discussed before, this by no means enough to capture all the intricacies of script's execution, but we should at least be able to rule out certain errors by defining a sufficiently strong precondition.

5.2.1 Propositions in a Dependently Typed Language

The *Curry-Howard Isomorphism* states that propositions correspond to types, and that proofs correspond to programs [13]. This means that for any proposition we can model as a type, we can prove that proposition by providing an inhabitant of that type.

$$\begin{array}{lcl}
true & \simeq & () \\
false & \simeq & \bot\ (Void) \\
P \land Q & \simeq & (P, Q) \\
P \lor Q & \simeq & Either\ P\ Q \\
P\ \Rightarrow\ Q & \simeq & P \to Q \\
lnotP & \simeq & P \to \bot\ (p \to Void) \\
\forall\, x : P\ x & \simeq & \{a : x\} \to P\ a \\
\exists\, x : P\ x & \simeq & \Sigma\ (x : A, P\ (x))
\end{array}$$

Sigma types are known as dependent pairs (*DPair*) in Idris. For convinience, the infix constructor >< is used in place of *DPair*. A value of type ($A >< P$) is constructed using $**$, e.g. ($value ** proof$).

5.2.2 Embedding of Predicates

A deep embedding exists for predicates in order to allow for easier manipulation of predicates, and more readable code. For example, consider the precondition of a simple script: `echo "Foo"; echo "Bar"`.

$$\mathit{true} \wedge (\forall\ (x : \mathit{String}) : \mathit{true} \wedge (\forall\ (y : \mathit{String}) : \mathit{true}))$$

This corresponds to the following type:

$$((), \{x : \mathit{String}\} \rightarrow ((), \{y : \mathit{String}\} \rightarrow ()))$$

Using a deep embedding for predicates, we can simply write:

$$[[..]]\ T \wedge (\mathit{Forall\ String}\ (\lambda x \Rightarrow T \wedge \mathit{Forall\ String}\ (\lambda y \Rightarrow T)))$$

Both expressions yield the same value, and are interchangeable. The `[[..]]` modifier is simply defined as a function with type $\mathit{Predicates} \rightarrow s \rightarrow \mathit{Type}$ that yields an appropriate type for a given predicate. The *Predicate* datatype in its entirety is defined as follows:

$$\begin{array}{l}
\textbf{data}\ \mathit{Predicate} : \mathit{Type} \rightarrow \mathit{Type}\ \textbf{where} \\
\quad (/\backslash) : \mathit{Predicate}\ s \rightarrow \mathit{Predicate}\ s \rightarrow \mathit{Predicate}\ s \\
\quad (:=>) : \mathit{Predicate}\ s \rightarrow \mathit{Predicate}\ s \rightarrow \mathit{Predicate}\ s \\
\quad \mathit{Forall} : (a : \mathit{Type}) \rightarrow (a \rightarrow \mathit{Predicate}\ s) \rightarrow \mathit{Predicate}\ s \\
\quad \mathit{Exists} : (a : \mathit{Type}) \rightarrow (a \rightarrow \mathit{Predicate}\ s) \rightarrow \mathit{Predicate}\ s \\
\quad \mathit{Atom} : (s \rightarrow \mathit{Type}) \rightarrow \mathit{Predicate}\ s \\
\quad T : \mathit{Predicate}\ s \\
\quad F : \mathit{Predicate}\ s
\end{array}$$

The *Atom* constructor allows for the inclusion of properties that reason about values of the type that a predicate ranges over. For example, suppose we are constructing a predicate that ranges over natural numbers and want to specify that a number is equal to some nuber m. We could use the following atomic predicate:

$$\begin{array}{l}
\mathit{isM} : \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Type} \\
\mathit{isM}\ m = [[..]]\ (\mathit{Atom}\ \$\ (\lambda n \Rightarrow n = m))
\end{array}$$

Obviously, we can only construct an inhabitant for $\mathit{isM}\ m\ n$ if $n = m$.

5.2.3 Provided Atomic Predicates

The approach for constructing atomic predicates described in the previous section can just as well be employed to define properties of other types. In our case, it makes most sense to define preconditions to be of the type $\mathit{Predicate}\ \mathit{FSTree}$ (i.e. a script's precondition ranges over the state of the filesystem). Atomic predicates to specify the following properties are provided:

- Whether a given path exists in the filesystem at all
- If a path exists, whether the node it points to is of a certain type (i.e. *File* or *Directory*)
- If a path exists, whether the current user has certain permissions on the node that the path points to.

The former two are quite easily specified once we have a datatype in place that holds proofs that a given path exists in a filesystem. For this datatype, we draw inspiration from the *Elem* datatype, which is a witness to the fact that a certain element can be found in a list. The resulting datatype is defined as follows:

```
data FSElem : Path → FSTree → Type where
  HereDir : FSElem (DirPath [])
                   (FSNode x xs)
  HereFile : (n1 = n2) → FSElem (FilePath [] n1)
                                (FSLeaf (MkFileInfo n2 md))
  ThereDir : (fs : FSTree) → FSElem (DirPath xs) fs
                           → Elem fs ys → (n : String)
                           → FSElem (DirPath (n :: xs))
                                    (FSNode (MkFileInfo n md) ys)
  ThereFile : (fs : FSTree) → FSElem (FilePath xs x) fs
                            → Elem fs ys → (n : String)
                            → FSElem (FilePath (n :: xs) x)
                                     (FSNode (MkFileInfo n md) ys)
```

Upon closer inspection we see that this definition closely follows the structure of the *FSTree* and *Path* datatypes. Any directory path with no components (i.e. "/") is part of a filesystem that has a node at the root. Any file with no components (i.e. "/filename.ext") is part of a filesystem that is just a leaf, provided the file in the leaf has the same name.

In the recursive case, a path is in a filesystem if the first component is equal to the name of the file that is at the root node of the filesystem, there is a proof that the remainder of the path is part of some other filesystem, and there is a proof that said filesystem is one of the children of the root node.

5.2.4 Constructing Atomic Proofs

Constructing values of the *FSElem* datatype is quite a lot of work, so a library function *provePathExists* is provided that takes care of this for the user. It has the following type signature:

$$\mathit{total}\ \mathit{provePathExists} : (p : \mathit{Path}) \to (\mathit{fs} : \mathit{FSTree}) \to \mathit{Dec}\ (\mathit{FSElem}\ p\ \mathit{fs})$$

Dec is an datatype from the Idris prelude representing decidable properties, and is equivalent to *Either* P $(P \to \mathit{Void})$. Hence the *provePathExists* function either provides a proof that the given path is part of the filesystem, or provides a proof of the contrary. Quant Deciding whether a path is part of a filesystem is quite easy for most cases. Only constructing a contra proof for recursive cases (i.e. if the input path is a nonempty component list and the input tree was constructed using *FSNode*) is a bit tricky.

For the recursive contra proof, we utilize the *Any* datatype, which can be found in `Data.List.Quantifiers`. A value of type *Any* P *xs* proves that there is at least one element of *xs* that satisfies P. Every child filesystem is mapped to a value of type *Either* P $(P \to \mathit{Void})$, and we construct a value of *Dec* (*Any isLeft xs*) which tells us whether any of the recursive values is a *Left* (and thus a proof). If the latter is not the case, we know that all recursive calls resulted in a contra proof. This allows us to construct a contra proof for the entire node. The actual result value of the recursive calls is a bit more complicated, but the structure remains broadly similar to the one described above.

As mentioned before, proving additional properties over the vertex pointed to by a path is quite trivial given a proof that the path exists. A way in which we could describe that the vertex pointed to by a path is of a certain type (i.e. a file or a directory) is with a dependent pair consisting of a value of type *FSElem p fs* and an equality proof that the object referenced by the proof has indeed a certain type. We use the following definitions for this. The functions *getFType* and *fileFromProof* respectively get the type from a file, and a file from a proof.

$$\begin{aligned}
&\mathit{total}\\
&\mathit{typeIs} : \mathit{FType} \to \mathit{FSElem}\ p\ \mathit{fs} \to \mathit{Type}\\
&\mathit{typeIs}\ \mathit{ft}\ \mathit{prf} = \mathit{getFType}\ (\mathit{fileFromProof}\ \mathit{prf}) = \mathit{ft}\\
&\mathit{total}\\
&\mathit{hasType} : (p : \mathit{Path}) \to (t : \mathit{FType}) \to (\mathit{fs} : \mathit{FSTree}) \to \mathit{Type}\\
&\mathit{hasType}\ p\ \mathit{ft}\ \mathit{fs} = \mathit{FSElem}\ p\ \mathit{fs} >< \mathit{typeIs}\ \mathit{ft}
\end{aligned}$$

Assuming *pathExists p fs* = *FSElem p fs*, we can now require files to be of a certain type in our preconditions. For example, a precondition for the *cat* command could be:

$$\begin{aligned}
\mathit{pre}\ (\mathit{Cat}\ p\ \mathit{cmd}) = {} & (\mathit{Atom}\ \$\ \mathit{pathExists}\ p)\\
& \wedge (\mathit{Atom}\ \$\ \mathit{hasType}\ p\ F\_)\\
& \wedge \mathit{Forall}\ \mathit{String}\ (\lambda \mathit{str} \Rightarrow \mathit{pre}\ (\mathit{cmd}\ \mathit{str}))
\end{aligned}$$

Construction a function *provePathHasType* now becomes an easy task:

$$\begin{aligned}&\mathit{provePathHasType} : (p : \mathit{Path}) \to (\mathit{ft} : \mathit{FType}) \to (\mathit{prf} : \mathit{FSElem}\ p\ \mathit{fs})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Dec}\ (\mathit{typeIs}\ \mathit{ft}\ \mathit{prf})\\ &\mathit{provePathHasType}\ p\ \mathit{ft}\ \mathit{prf} = \mathit{decEq}\ (\mathit{getFType}\ \$\ \mathit{fileFromProof}\ \mathit{prf})\ \mathit{ft}\end{aligned}$$

6 Implementation

The implementation of the project described in this report can be found on GitHub [4]. Three approaches have been tried, their code can be found in the corresponding directory.

6.1 Shallow Embedding Using Control.ST

A first attempt towards safer shell scripts was made using the *Control.ST* library (found in the *contrib*) package. A description and motivation of the library's design and implementation is described in the paper *State Machines All The Way Down* by Edwin Brady [5].

As implied by the title of the accompanying paper, the *Control.ST* library centers around the idea of state machines, where states carry a collection of associated resources. The *STrans* type describes how resources change when a function is invoked, i.e. which resources are required as input, and which remain (or are created) after the function is run. This structure becomes clear when considering the *STrans* type:

$$\begin{aligned}&\mathit{STrans} : (m : \mathit{Type} \to \mathit{Type}) \to\\ &\quad (\mathit{resultType} : \mathit{Type}) \to\\ &\quad (\mathit{in\_res} : \mathit{Resources}) \to\\ &\quad (\mathit{out\_res} : \mathit{resultType} \to \mathit{Resources}) \to\\ &\quad \mathit{Type}\end{aligned}$$

As can be seen, *STrans* is dependent, in the sense that the collection of resulting resources is determined by the function's result. A function resetting some integer resource to zero might look like this:

$$\begin{aligned}&\mathit{reset} : (x : \mathit{Var}) \to \mathit{STrans}\ m\ ()\ [\,]\ [\mathit{const}\ [x ::: \mathit{State}\ \mathit{Int}]]\\ &\mathit{reset}\ x = \mathit{write}\ x\ 0\end{aligned}$$

The same concept can be applied to enforce that a script obeys its a contract, simply by tracking resources describing capabilities. Assuming a datatype for cababilities that contains a path and the kind of authority required (read or write), we can write a function that uses the *cat* command.

$$
\begin{aligned}
&\mathit{myScript} : (\mathit{path} : \mathit{Path}) \to \{\,\mathit{contract} : \mathit{Var}\,\}\\
&\qquad\qquad\qquad\qquad\quad \to \mathit{ST}\ \mathit{IO}\ ()\ [\,\mathit{contract} ::: \mathit{Composite}\ [\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Require}\ (\mathit{MkCapability}\ \mathit{path}\ R)\\
&\qquad\qquad\qquad\qquad\qquad]]\\
&\mathit{myScript}\ \mathit{path} = \mathbf{do}\\
&\quad \mathit{call}\ (\mathit{cat}\ \mathit{path})
\end{aligned}
$$

The *cat* function has type signature $(\mathit{pat} : \mathit{Path}) \to \mathit{STrans}\ m\ ()\ [\,\mathit{contract} ::: \mathit{Composite}\ [\mathit{Require}\ (\mathit{MkCapability}\ \mathit{path}\ R)]]$, requiring a capability with read authority over its input path. Hence, we can only write a function that uses *cat* if the resources required by *cat* are part of its contract. Though the *Control.ST* library looks promising for our purposes, there are a few disadvantages to consider. First, it is a shallow embedding. This means that there is no real separation between syntax and semantics, meaning that we cannot easily change the interpretation of a script, making it harder to for example conduct tests with mock filesystems. This issue could be partly resolved through an interface constraint on the context in which the script runs, but this is not ideal. Furthermore, when looking at the library's documentation [9], we see that it is often necessary to explicitly pass around references to resources, which might significantly pollute the code in more complex use-cases. Due to these reasons we decided to pursue a different solution.

6.2 The HoareState Monad

To properly capture dependencies between sequenced commands, we turn to something called the *HoareState* monad [**?**]. First, let us recall the definition of the regular state monad:

$$
\begin{aligned}
&\mathit{State} : \mathit{Type} \to \mathit{Type} \to \mathit{Type}\\
&\mathit{State}\ s\ a = s \to (a, s)
\end{aligned}
$$

and its accompanying bind operation:

$$
\begin{aligned}
&(\gg\!=) : \mathit{State}\ s\ a \to (a \to \mathit{State}\ s\ b) \to \mathit{State}\ s\ b\\
&f \gg\!= g = \mathit{uncurry}\ g \circ f
\end{aligned}
$$

The *HoareState* monad is embellishes this definition with a pre- and postcondition ranging over respectively the in- and output state.

6.2.1 Definition of the HoareState Monad

We start by defining the *HoareState* type. The precondition simply maps the input state to a type representing the desired proposition. The postcondition is similar, though the resulting type may not only depend on the output state, but also on the input state and the resulting value.

$$
\begin{aligned}
&\mathit{HoareState} : (a : \mathit{Type}) \to (s : \mathit{Type}) \to (s \to \mathit{Type}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to (s \to a \to s \to \mathit{Type}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Type} \\
&\mathit{HoareState}\ s\ a\ \mathit{pre}\ \mathit{post} = \\
&\quad (i : s >< \mathit{pre}) \to (a, s) >< \mathit{post}\ (\mathit{fst}\ i)
\end{aligned}
$$

A *bind* operation for the *HoareState* monad can be obtained by observing that for every $f \gg\!= g$, $\mathit{pre}\ f \Rightarrow \mathit{post}\ g$ should hold. Furthermore, state and result value such that $\mathit{post}\ f$ and $\mathit{pre}\ g$ hold should exist. In human language, this means that it should be possible to come up with an intermediate state and result value such that both the postcondition of the first computation and the precondition of the second computation are satisfied. This gives rise to the following type definition for $\gg\!=$, inspired by the Agda implementation found in [8]:

$$
\begin{aligned}
&(\gg\!=) : \{\mathit{p1} : s \to \mathit{Type}\} \to \{\mathit{q1} : s \to a \to s \to \mathit{Type}\} \to \\
&\qquad \{\mathit{p2} : a \to s \to \mathit{Type}\} \to \\
&\qquad \{\mathit{q2} : a \to s \to a \to s \to \mathit{Type}\} \to \mathit{HoareState}\ s\ a\ \mathit{p1}\ \mathit{q1} \to \\
&\qquad ((x : a) \to \mathit{HoareState}\ s\ a\ (\mathit{p2}\ x)\ (\mathit{q2}\ x)) \to \\
&\qquad \mathit{HoareState}\ s\ b\ (\lambda \mathit{s1} \Rightarrow \\
&\qquad\quad (\mathit{p1}\ \mathit{s1}, ((x : a) \to (\mathit{s2} : s) \to \mathit{q1}\ \mathit{s1}\ (x, \mathit{s2}) \to \mathit{p2}\ x\ \mathit{s2})) \\
&\qquad)\ (\lambda \mathit{s1}, (x, \mathit{s3}) \Rightarrow \\
&\qquad\quad ((a, s) >< (\lambda(y, \mathit{s2}) \Rightarrow (\mathit{q1}\ \mathit{s1}\ (y, \mathit{s2}), \mathit{q2}\ y\ \mathit{s2}\ (x, \mathit{s3}))))) \\
&f \gg\!= g = \lambda(\mathit{s1} ** \mathit{prf}) \to \\
&\quad \textbf{case}\ f\ (\mathit{s1} ** (\mathit{fst}\ \mathit{prf}))\ \textbf{of} \\
&\qquad ((x,\ \mathit{s2}) ** p) \Rightarrow \\
&\qquad\quad \textbf{case}\ g\ x\ (\mathit{s2} ** ((\mathit{snd}\ \mathit{prf})\ x\ \mathit{s2}\ p))\ \textbf{of} \\
&\qquad\qquad ((y, \mathit{s3}) ** q) \Rightarrow ((y, \mathit{s3}) ** ((x, \mathit{s2}) ** (p, q)))
\end{aligned}
$$

It is important to realize that the input proof of the lambda expression is a value that inhabits the aggregated precondition. Once we know this, it is easy to see that we have all the ingredients to construct a sensible definition.

6.2.2 The HoareState Monad in Idris

With a suitable definition for $\gg\!=$ in place, we can define some basic operations for the *HoareState* monad. We assume that $\mathit{Top} = \mathit{const}\ \mathit{Unit}$, i.e. always *true*.

$$
\begin{aligned}
&\mathit{return} : (x : a) \to \mathit{HoareState}\ s\ a\ \mathit{Top}\ (\lambda \mathit{s1}, y, \mathit{s2} \Rightarrow (\mathit{s1} = \mathit{s2}, y = x)) \\
&\mathit{return}\ x\ (s ** \_) = ((x, s), (\mathit{Refl}, \mathit{Refl}))
\end{aligned}
$$

```
get : HoareState s s Top (λs1, x, s2 ⇒ (s1 = s2, x = s2))
get (s ** _) = ((s, s), (Refl, Refl))
put : (x : s) → HoareState s () Top (\_, _, s2 ⇒ x = s2)
put x _ = ((), x), Refl)
```

However, when attempting to write small programs with these definitions, we encounter some difficulties. For example, the following program is not accepted by the typechecker.

```
program : HoareState Int Int
  (λs ⇒ ((), Unit → Int → Int → ()))
    (λs1 ⇒ λv ⇒ ((a, s) ><
      (λ(y, s2) ⇒ (s2 = 10, (s2 = snd v, snd v = fst v)))))
program = hput 10 `hbind` (\_ ⇒ hget)
```

When compiling the above code, we are met with the following error message:

```
Type mismatch between
    case v0 of
      (x, s3) ⇒ (a, s) ><
        (λlamp ⇒
          case lamp of
            (y, s2) ⇒ (q1 s1 (y, s2), q2 y s2 (x, s3)))
  and
    λv ⇒
      ({lamp_0} : (a, s) **
      case lamp of
        (y, s2) ⇒ (s2 = 10, (s2 = snd v, snd v = fst v)))
```

Ignoring the myriad of auxiliary variables created internally by Idris, we see that the typechecker unfortunately rejects this definition, because it deems that $(q1\ \ s1\ \ (y, s2), q2\ \ y\ \ s2\ \ (x, s3))$ is not equal to $(s2 = snd\ \ v, snd\ \ v = fst\ \ v)))$. Based on our understanding of the concerning pre- and postconditions, there is no clear reason why this is the case. It might be the case that Idris's definition for type equality simply is not strong enough for this case.

The error message above is merely an example of the many seemingly unexplainable errors that were encountered. Despite all its merits, Idris is still a language under development. This shows in the confusing error messages, and the fact that whether the typechecker accepts your code sometimes depends on how exactly you write a definition (e.g. pattern matching on the arguments of a lambda expression results in an error, while using a case expression is accepted).

Adding in sparse documentation and a programmer relatively inexperienced with dependent types, it is probably best to admit that attempting to make the *HoareState* monad work for our purposes is not the best way forward, and that it might be better to proceed with a slightly simpler approach.

6.2.3 Precondition Strengthening and Postcondition Weakening

Hoare Logic gives us the possibility to *strengthen* a precondition and *weaken* a postcondition [15]. Given a hoare tripple $\{P\}\ c\ \{Q\}$, we may say that:

$$\{P\}\ c\ \{Q\} \land P' \Rightarrow P \implies \{P'\}\ c\ \{Q\}$$
$$\{P\}\ c\ \{Q\} \land Q \Rightarrow Q' \implies \{P'\}\ c\ \{Q'\}$$

Similarly, if we have a function that transforms a value of type *p2* into a value of type *p1*, we can strengthen the precondition of a *HoareState* with the following function:

```
strengthen : { a : Type } → { s : Type } → { q : Post s a }
                          → { p1 : Pre s } → { p2 : Pre s }
                          → ((i : s) → p2 i → p1 i)
                          → HoareState s a p1 q
                          → HoareState s a p2 q
strengthen f (HS st) (i ** p) = st (i ** (f i p))
```

In a similar fashion a function *weaken* can be defined that can transform values of type *HoareState s a p q1* into *HoareState s a p q2* if given a function with type *q1* → *q2*. These functions are mostly beneficial when writing shell scripts intended for reuse. The pre- and postconditions assembled by ≫= can become complicated very quickly and they can be greatly simplified by using appropriate rewrites. On the other hand, we may circumvent some of the issues described in the previous section by rewriting pre- and postconditions, although given a function *p2* → *p1* used for precondition strengthening, the typechecker would still need to recognize that *p2* is equal to the assembled precondition that we want to simplify.

6.3 Free Monads

Parallel to the work on the *HoareState* monad, focussed shifted from a using shallow embedding of shell commands to using a more deep embedding. Preferably, such an embedding would allow for relatively easy extension of the set of commands, while simultaneously seperating syntax from semantics.

6.3.1 Syntactical Definition of the Scripting Language

Creating a monads for free from arbitrary functors, Free monads allow for such seperation [14]. Consider the definition of the *Free* datatype:

```
data Free : (Type → Type) → Type → Type where
  Bind : Functor f ⇒ f (Free f a) → Free f a
  Pure : Functor f ⇒ a → Free f a
```

Additionally, we need a suitable datatype to represent our shell commands:

```
data Cmd next = Ls Path (List Path → next)
              | Cat Path (String → next)
              | Echo String (String → next)
              | Return
```

In general, the structure of each constructor is similar, with fields for input parameters, and a continuation. Only the *Return* constructor is different, signifying the end of a sequence. We define the following *Monad* instance for the *Free* datatype:

```
implementation Functor f ⇒ Monad (Free f) where
  f ≫= g = assert_total $
    case f of
      (Pure m) ⇒ g m
      (Bind m) ⇒ Bind (map (≫=g) m)
```

Using a *liftF* function that lifts a value of some functor into the *Free* datatype, we can define smart constructors for shell commands:

```
liftF : Functor f ⇒ f a → Free f a
liftF m = Bind (map Pure m)

done : a → Free Cmd a
done x = liftF (Return)

ls : Path → Free Cmd (List Path)
ls path = liftF (Ls path id)

cat : Path → Free Cmd String
cat path = liftF (Cat path id)

echo : String → Free Cmd String
echo str = liftF (Echo str id)
```

This enables the programmer to assemble shell scripts using *do* notation (or any other tool from the monadic toolkit for that matter). An added benefit of this approach is that our shell scripts are automatically typesafe. For example, attempting to compile something like *cat* (*echo* `"Hello, World!"`) results in a type error; after all, *echo* yields a *String* while *cat* expects a *Path*. Also notice that ≫= can be used to pipe results between commands, allowing us to write something that is syntactically surprisingly similar to actual shell scripts, with the added benefit that all the commands are now typed.

```
program : Free Cmd ()
program = do
  ls (DirPath ["somedir"])
  echo "Hello, World" >>= echo
  done ()
```

6.3.2 Function Totality

It is important to make a few remarks about function totality befor proceeding to how to run our scripts. Contrary to Agda, functions are not required to be total in Idris. It is however possible to mark functions as total, and the compiler will run a totality check to try to prove totality for the those functions. Furthermore, functions that appear in type signatures will only be expanded if they are known to be total (i.e. the totality checker can prove that the function is total). This is in order to guarantee termination of the typechecker.

As we will see, a value of type *pre script* is needed as a proof of the script's precondition in order to run it. Here *pre* is simply a function that assembles a precondition for a value of type *Free Cmd a*. Since *pre* will occur as part of a type signature, it needs to be total. If that were not the case, the typechecker would never be able to decide whether the provided *check* function actually tries to prove the correct precondition. This leads to the following definition for *pre*:

```
total pre : Free Cmd a -> Predicate FSTree
pre (Bind cmd) = assert_total $
  case cmd of
    (Ls p cmd) => (
      Atom $ pathExists p) /\ Forall (List Path) (\lst => pre (cmd lst))
    (Cat p cmd) =>
      (Atom $ pathExists p) /\ (Atom $ hasType p F_) /\
        Forall String (\str => pre (cmd str))
    (Echo s cmd) =>
      Forall String (\str => pre (cmd str))
    Return => T
pre (Pure _) = T
```

This definition, however, is not accepted by the totality checker; we get the following error message: `pre is possibly not total due to: Free.Bind`. This is a problem because we need the precondition of a script to be expanded by the typechecker!

The cause of this problem is that the totality checker cannot know for sure that *Free f a* is strictly positive, since this depends on whether its functor, *f*, is strictly positive. Of course we know that *Free Cmd a* is strictly positive, based on our knowledge of the *Cmd* datatype. However, there is no easy way for constraining the argument *f* of *Free f a* to strictly positive datatypes only. This issue can be partly circumvented by defining a separate datatype that exhibits the same structure as *Free Cmd a*:

```
data CmdF : Type → Type where
  Bind : Cmd (CmdF a) → CmdF a
  Pure : a → CmdF a
```

By defining *pre* over the *CmdF* datatype, which can be done by simply exchanging *Free Cmd a* with *CmdF a* in the type signature, the totality checker is able to recognize that *Bind* is strictly positive. This solution is not ideal however, since we now no longer work with the *Free* datatype. This means that we lose quite a bit of generality; functions can no longer be defined over the *Free* datatype, but have to be explicitly defined for the *CmdF* datatype.

A possible solution could be to somehow define a universe that captures strictly positive types. We could then define the *Cmd* type as a member of such a universe, and modify the *Free* datatype to work with such types (though we would still need to be able to impose a *Functor* constraint). There is some work in this direction [7], but there is really no way of telling how well it applies to our problem.

6.3.3 Running Shell Scripts

In order to execute shell scripts, we require a separate *run* function that takes a value of type *Free Cmd a* and produces a result. The *run* function should have roughly the following workflow: get an abstract representation of the filesystem, see if the precondition holds for that filesystem, and if so, proceed with execution of the script. Since there is not possible to define a general function that decides if a precondition holds or not, this proof obligation is shifted to the programmer. A *check* function needs to be supplied that yields a value of type *Maybe* (*pre script*)

```
run : (CmdExec m, Throwable m) ⇒
  (script : CmdF r) → ((fs : FSTree)
                        → Maybe (([[..]] (pre script)) fs)) → m ()
run script check = do
```

```
    fs ← getFS
    case check fs of
      Nothing ⇒ throw "Precondition check failed ..."
      (Just x) ⇒ cmdExec script
```

Notice that the *run* function is polymorphic in the context in which the input script is run, so long implementations the *CmdExec* interface (describing how to execute commands) and the *Throwable* interface (describing how to throw an error) are supplied. By leaving the exact context ambigious, we separate the actual implementation of the commands from the mechanics surrounding the precondition. This allows for scripts to execute in a context other than the *IO* monad, which might be useful for testing purposes.

The second argument is a function that determines whether the precondition of the script holds and yields a proof if that is the case. Execution of the script proceeds only if such a proof can be supplied. Below is the definition of the *CmdExec* interface:

```
interface Monad m ⇒ CmdExec (m : Type → Type) where
  cmdExec : Free Cmd a → m ()
    argc : f a → Nat
  inTypes : (inh : f a) → Vect (argc inh) Type
  outType : f a → Maybe Type
  getParse : (inh : f a) → String →
                 Either String (fromMaybe Unit (outType inh))
  exec : (inh : f a) → HVect (inTypes inh) → f String
  getParams : (inh : f a) → HVect (inTypes inh)
  getF : (inh : f a) → Either String (fromMaybe Unit (outType inh) → a)
```

The various functions describe the in- and output types of commands and how to parse their in- and output values. The *exec* function does the heavy lifting and actually executes the commands. A default implementation for *cmdExec* is supplied:

```
cmdExec : CmdExec m ⇒ Free Cmd a → m ()
cmdExec (Pure x) = pure ()
cmdExec (Bind cmd) = do
  output_raw ← exec cmd (getParams cmd)
  print output_raw
  fromRight (pure ())
    (do f ← getF cmd
      p ← getParse cmd output_raw
```

```
        pure $ (cmdExec (f p))
    )
```

This approach is rather verbose, and given the same overall structure shared by all constructors, it is probably possible to define most functions in the *CmdExec* interface over pattern functors, rather than as part of an interface.

An example implementation of the *CmdExec* interface is supplied for the *IO* monad, which theoretecally should allow us to write a shell script using smart constructors, prove its precondition, and compile an executable which we can run to execute the script. Take the following scrip:

```
echo1 : Free Cmd ()
echo1 = do
  echo "Foo" >>= echo
  pure ()
```

We create a function that calculates a proof of its precondition:

```
proveEcho1 : (fs : FSTree) → Maybe (([[..]] (
  Forall String (\_ ⇒
    Forall String (\_ ⇒ T)
  )))fs)
proveEcho1 _ = Just $ (const (const ()))
```

We can now create an Idris program that proves and executes the script *echo1*:

```
main : IO ()
main = run echo1 proveEcho1
```

Compilation proceeds with `idris -p contrib -p lightyear Main.idr -o script`. This yields the following output:

```
idris: Erasure/getArity: definition not found for with block in
        errorPrelude.Strings.strM
CallStack (from HasCallStack): error, called at
        src/Idris/Erasure.hs:605:20
                in idris-1.3.1-HTrT6RZ35FuzHOycTuJOO:Idris.Erasure
```

A quick google search reveals that the mentioned file deals with code generation, so unfortunately we cannot run our script right now, and since the typechecker has no complaints, there are unfortunately not really any pointers as to how we can prevent this from happening. It thus seems that we cannot actually run our proven scripts for the time being until this issue is resolved.

6.4 Expressivity

It might be interesting to consider what can and cannot be expressed using this approach. Intuitively, we might say that a pre- and postcondition are sufficient, as long as commands do not depend on the result of other commands. This begs the question of what dependencies mean in our context. Let us say that $mod(c)$ denotes the set of resources that is touched (i.e. added, removed or modified) by a command c, and $addr(P)$ denotes the set of resources referenced in a predicate P. We can then use the *frame rule* found in separation logic [12] to formalize a notion of independence:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}} mod(c) \cap addr(R) = \emptyset$$

If $\{P_1\}\ c_1\ \{Q_1\}$ and $\{P_2\}\ c_2\ \{Q_2\}$ are Hoare tripples, we can say that c_1 and c_2 are independent if we can find derivations for both $\{P_1 * (P_2 \wedge Q_2)\}\ c_1\ \{Q_1 * (P_2 \wedge Q_2)\}$ and $\{P_2 * (P_1 \wedge Q_1)\}\ c_2\ \{Q_2 * (P_1 \wedge Q_1)\}$. A script C is then entirely independent if we can find the aforementioned derivation for any $c_1, c_2 \in C$. A logical consequence of a script being completely independent is that we can freely reorder its commands, or execute them in parallel without affecting the outcome.

7 Conclusion

This report discusses various approach that may aid in bettering the safety of shell scripts by means of embedding in a host language with a dependent type system. Our language of choice is Idris, a dependently typed language that aims to unify the benefits of a strong type system with the ability to apply the language to actual programming problems. Though it is in many aspects very suitable for our cause, it might be interesting to see what is feasable in Agda or Haskell.

Various approaches have been tried, most notably a shallow embedding using the *Control.ST* library, automatic assembly of pre- and postconditions using the *HoareState* monad and finally settling on an embedding using *Free monads*. Although not all explorations turned out to be equally sucessful, each iteration contributed towards the final product in the form of newly aquired insights. This final product is probably best regarded as a proof of concept, showing that we can apply techniques from the realm of functional programming to solve problems in other domains.

Finally, it is important to note that the approaches discussed in this report are applicable beyond shell scripts. We might just as well prove properties over C programs and a heap; the same workflow and embedding techniques still apply.

8 Future Work

The approach presented clearly uses a lot of simplification, assumptions and abstractions, meaning that there are plenty of opportunities for future extensions. This section provides a small overview of some of the possibilities.

8.1 Syntax Extensions using Functor Coproducts

Since both the *run* as well as the *pre* functions are defined over *Free Cmd a* in favor of *Cmd a*, we may benefit from the extensibility of Free monads. *Data types à la carte* [10] describes how functions can be defined over Free monads derived from coproducts of functors. Although the paper uses Haskell as its implementation language, we can adapt the concepts outlined to Idris. In the paper, Haskell's typeclass system is used to describe how a functor can be injected into some coproduct:

```
class (Functor sub, Functor sup) ⇒ sub :<: sup where
  inj :: sub a → sup a
instance Functor f ⇒ f :<: f where
  inj = idprovide static guaranteesprovide static guarantees
instance (Functor f, Functor g) ⇒ f :<: (f : + : g) where
  inj = Inl
instance (Functor f, Functor g, Functor h, f :<: g) ⇒
  f :<: (h : + : g) where
  inj = Inr ∘ inj
```

The second and third instance overlap, which is problematic since Idris does not really have a system in place to deal with overlapping interfaces. We can resolve this problem by using a datatype in favor of an interface:

```
data (:<:) : (Type → Type) → (Type → Type) → Type where
  Ref : (Functor f) ⇒ f :<: f
  Co  : (Functor f, Functor g) ⇒ f :<: (f : + : g)
  Ind : (Functor f, Functor g, Functor h) ⇒
          {auto prf : f :<: g} → f :<: h : + : g
```

and have the *inj* function describe how values of this datatype can be combined

```
inj : {p : f :<: g} → f a → g a
inj {p} x with (p)
  inj x | Ref         = x
```

$$
\begin{aligned}
&\mathit{inj}\ x \mid \mathit{Co} && = \mathit{Inl}\ x \\
&\mathit{inj}\ x \mid \mathit{Ind}\ \{\mathit{prf}\} && = \mathit{Inr}\ (\mathit{inj}\ \{p = \mathit{prf}\}\ x)
\end{aligned}
$$

We can then define the *inject* function using Idris's *auto* keyword, effectively replacing the resolving of Typeclass constraints with an proof search that tries to construct a term of type f :<: g. Assuming *inj* is the only function in scope that modifies terms of type f :<: g, these are equivalent.

$$
\begin{aligned}
&\mathit{inject} : (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \\
&\qquad \{\mathit{auto}\ \mathit{prf} : g :<: f\} \rightarrow g\ (\mathit{Free}\ f\ a) \rightarrow \mathit{Free}\ f\ a \\
&\mathit{inject}\ \{\mathit{prf}\} = \mathit{Bind} \circ (\mathit{inj}\ \{p = \mathit{prf}\})
\end{aligned}
$$

All other function definitions remain virtually identical. Since the solution provided in the paper also retains the separation between syntax and interpretation, converting the *pre* and *cmdExec* functions to algebras over coproducts seems rather straightforward. Oviously, the previously described problems with function totality remain, and simply applying the conversion described here is not sufficient. We would still need to find a way to constraint ourselves to strictly positive datatypes only, though it is easy to argue that if f and g are both strictly positive, so is $f : + : g$.

8.2 Shell Features

There are many shell features that are not supported by the model presented in this report. A few that that may be interesting extensions are listed below.

8.2.1 Control Flow

Currently, scripts are presented as a mere sequence of commands. However, many shells support more complex control flow in the form of `if` and `while` statements. Seeing how these control flow structures can be incorporated into our embedding can be an interesting extension. Looking at Hoare logic, we see that `if` statements are rather straightforward. Loops, however, will probably prove to be quite a bit more challenging, as they often require annotation with an invariant to properly calculate a precondition.

8.2.2 Command Line Options

The behaviour of commands can often be altered by passing different parameters/flags. Currently we cannot model this dependency between input and behaviour/output.

8.2.3 File Contents

Real shell scripts have the option to use the contents of a file freely as input. In our model this is of course also possible, but since *cat* is said to return a *String*, we would need explicit parsing to interpret it as a value of any other type. Additionally it is of course not possible to proper reason about these kind of scripts, since our model of the filesystem omits the file's content.

References

[1] Moore, S., Dimoulas, C., King, D., & Chong, S. (2014, October). SHILL: A Secure Shell Scripting Language. In OSDI (pp. 183-199).

[2] Brady, E. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming, 23(5), 552-593.

[3] Saltzer, J. H. (1974). Protection and the control of information sharing in Multics. Communications of the ACM, 17(7), 388-402.

[4] GitHub repository with projet codebase. Retrieved from https://github.com/casvdrest/ep-idris

[5] Brady, E. (2016). State Machines All The Way Down.

[6] Swierstra, W. (2009, August). A Hoare logic for the state monad. In International Conference on Theorem Proving in Higher Order Logics (pp. 440-451). Springer, Berlin, Heidelberg.

[7] Abbott, M., Altenkirch, T., & Ghani, N. (2005). Containers: constructing strictly positive types. Theoretical Computer Science, 342(1), 3-27.

[8] Swierstra, W. (2009). A functional specification of effects (Doctoral dissertation, University of Nottingham).

[9] Control.ST documentation. (n.d.). Retrieved from http://docs.idris-lang.org/en/latest/st/introduction.html

[10] Swierstra, W. (2008). Data types à la carte. Journal of functional programming, 18(4), 423-436.

[11] Swierstra, W., & Löh, A. (2014, October). The semantics of version control. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (pp. 43-54). ACM.

[12] Wadler, Philip. "Propositions as types." Communications of the ACM 58.12 (2015): 75-84.

[13] Gonzalez, G. (2012, June 09). Haskell for all. Retrieved from http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html

[14] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. Communications of the ACM, 12(10), 576-580.