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 that is preferably avoided: it is cumbersome at best, and realistically unfeasible in many cases. Furthermore, a script is often executed with much 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.

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.

In this project, I have attempted to provide a solution that mitigates these issues by embedding a small subset of Bash into Idris[2], utilizing its dependent type system to model a script's behaviour, and statically enforce that the claims made by a script are respected.

2 Related Work

The approach taken in the project is largely based on *Shill*[1], a scripting language developed at Harvard University. Shill is based around the *principle of least privilege* (a script should have no more authority than it strictly needs), and takes a sanbox-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 sanbox, and thus are restricted in the same way.

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

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

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 that could utilize the above contract is:

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

Although Shill's API provides the necessary tools to specify fine-grained authority for scripts, all enforcement of contracts happens dynamically. This comes with the obvious downside of how to deal with scripts that fail dynamically halfway through their execution. Preferably we would deal with this scenario by preventing scripts that violate their contract from executing at all!

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 and/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:

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

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
```

The `touch` command does not care whether *file.txt* already exists, but `cat` fails in that case. However, requiring that *file.txt* exists makes the precondition to strong; a successful execution of `touch file.txt` guarantees that *file.txt* exists when we arrive at the `cat` statement.

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 is assumed to have a fixed set of parameters and return type.

3.2 File System

4 Implementation

4.1 Filesystem

In order to reason about the effects of a command on a filesystem, we need some kind of abstract representation. 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 metadata of that particular vertex, including permissions, name and the type (file or directory). The contents of a file are not recorded.

This results in the following datatype definition:

```
data FSTree = FSNode FileInfo (List FSTree)  -- Directories
            | FSLeaf FileInfo                -- Files
```

It should be obvious that leafs are only meant to contain files, and nodes are supposed to contain directories.

4.2 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 behaviour, but we should at least be able to rule out certain errors by defining a sufficiently strong precondition.

4.2.1 Embedding of Predicates

In a dependently typed language, propositions are commonly defined as a type, and justified by supplying a definition that inhabits said type. Converting a formula in predicate logic to its corresponding type is relatively straightforward. I assume the following mapping:

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

Sigma types are modelled as dependent pairs 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)`.

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 two subsequent `echo` commands:

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

This corresponds to the following type:

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

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

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

Both expressions yield the same value, and are interchangeable. The `[[..]]` modifier is simply defined as a function with type *Predicates* $\to s \to$ *Type* that yields an appropriate type for a given predicate.

The *Predicate* datatype in its entirety is defined as follows:

```
data Predicate : Type → Type where
  data Predicate : Type → Type where
  (/\) : Predicate s → Predicate s → Predicate s
  (:=>) : Predicate s → Predicate s → Predicate s
  Forall : (a : Type) → (a → Predicate s) → Predicate s
  Exists : (a : Type) → (a → Predicate s) → Predicate s
  Atom : (s → Type) → Predicate s
  T : Predicate s
  F : Predicate s
```

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:

```
isM : Nat → Nat → Type
isM m = [[..]] (Atom $ (λn ⇒ n = m))
```

Obviously, we can only construct an inhabitant for *isM m n* if $n = m$.

4.2.2 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 *Predicate 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)
```

Though rather lengthy, the definition is actually quite straightforward. 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.

4.3 Shallow Embedding Using Control.ST

4.4 HoareState

4.5 Free Monads

5 Conclusion

6 Future Work

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] Krohn, M. N., Efstathopoulos, P., Frey, C., Kaashoek, M. F., Kohler, E., Mazieres, D., ... & Ziegler, D. (2005, June). Make Least Privilege a Right (Not a Privilege). In HotOS.