Embedding a Firewall Programming Language into Haskell

André Santos
alms@di.ufpe.br
Departamento de Informática - UFPE

Abstract

In this paper we present how a domain-specific language, a language to program network firewalls, was embedded into the Haskell functional language.

The resulting *embedded* language gives systems administrators a language similar to the one they currently use, but with a richer syntax and semantics, with many advantages over the original environment, such as the flexibility to extend and improve definitions of rules and constants. Also, there is no need for the systems administrator to learn Haskell, or even to realise that he or she is using Haskell.

1 Introduction

Domain-specific languages (DSLs) are present in a many areas in computing. They are used for programming (or configuring) specific equipments (like routers and switches), for preparing documents (like the TEX/LATEX typesetting languages), for text processing (like `sed` and `awk`), specifying lexers and parsers (`lex` and `yacc`) etc.

Domain-specific languages are designed with the specific intent of having a syntax and semantics that is easy to understand and use by the people that work on that domain, instead of *forcing* them to program in a more generic or powerful language. Actually, not *allowing* the user to step into more elaborate programming might be one of the goals of the language.

Although there are advantages in having domain-specific languages, their development sometimes might be just as difficult as a generic programming language: one not only has to decide on a syntax and semantics, but also has to write parsers, (possibly) type checkers and code generators. This can be a very time-consuming process.

An approach that has recently been used to allow a quicker development and experimentation with DSLs is to *embed* the language into a *host* language [Hud96]. But not all languages are suitable for providing (almost) transparently a good syntax and semantics for another language. This approach has shown some success with the use of the lazy functional programming language Haskell [HPW$^+$92] as the *host* language. Some such examples are the Haskore language [HMGW96], used to write (i.e. *to program*) music, the Functional Reactive Animation (Fran) library [Ell97], used to program graphics animations, and Hawk [MLC98], used to specify and simulate processsors.

In the next sections we present how we embedded a DSL for router security configuration into Haskell, and the advantages of this approach.

2 Router Security Programming

Router programming languages vary according to the router manufacturer, the version of the router operating system, and to the features available in the router. Most of the configuration of a router is rarely modified, and probably wouldn't justify writing a separate DSL for programming it. But many routers can be used as a simple and efficient firewall. Firewalls essentially provide security to a site by filtering TCP/IP packets from coming into the site, and therefore denying access to many services and hosts from the outside world (e.g. disk sharing by NFS shouldn't be available outside the company). Through the use of rules for filtering packets, one can restrict a certain traffic only to specific hosts, and therefore only has to worry about the external security of that service on those hosts (e.g. external telnet service is only allowed to a specific host). These filters are changed much more often by the site administrator, since as soon as a service moves from a host or a new service is offered to the outside world these filters must be changed.

We will concentrate on the language provided by Cisco routers (Cisco IOS operating system), but routers from other manufacturers are very similar. The commands always start with the number of the access-list, which defines a filtering rule, permitting or denying a connection. The rules are specified as follows:

- first, the `permit` or `deny` keyword;
- then the protocol is identified, e.g. tcp or udp;
- then the source of the packet is specified, either by a pair consisting of an IP number and an IP mask[1], by the `host` keyword followed by the IP number of the host, or by the keyword `any`, which means "any host".
- the destination of the packet, specified in the same way as the source.
- then we have to specify the service that is being allowed or denied (or the port numbers used by the service, if it is not in a list of predefined services). This is done by specifying that a match happens if the target port number of the packet is "equal to" (eq) a given port number (or service name).

For example, the rule:

```
access-list 101 permit tcp any host 150.161.2.50 eq www
```

allows tcp connections to the www/http port (port 80) from any host to the host 150.161.2.50.

The sample access-list below presents some of the difficulties a systems administrator faces when programming access-lists[2]:

[1] The mask identifies if the IP number is a host or a network.

[2] lines starting with exclamation marks are comments.

```
access-list 101 permit tcp any any established
access-list 101 permit tcp any host 150.161.2.1 eq www
access-list 101 permit tcp any host 150.161.2.202 eq www
! IMAP (mail) protocol uses ports 143 and 220 and both tcp and udp:
access-list 101 permit tcp any host 150.161.2.1 eq 143
access-list 101 permit tcp any host 150.161.2.1 eq 220
access-list 101 permit udp any host 150.161.2.1 eq 143
access-list 101 permit udp any host 150.161.2.1 eq 220
! ssh protocol is not predefined, port 22
access-list 101 permit tcp any host 150.161.2.50 eq 22
access-list 101 deny ip any any
```

From the example above, we can notice that there are some inconveniences with the notation:

- It is not enough to know which service you want to offer, you have to know wether it uses tcp and/or udp;
- In older versions of the Cisco software you have to know the port numbers of the service you want to restrict. Even in more recent versions, where you can use a *service name*, the set of names available is fixed and cannot be extended, therefore one still has to resort to using port numbers in many cases;
- the specification of hosts in the old notation with an IP-mask pair is also prone to error and confusion;
- the syntax has no notion of scope or structuring, as can be seen by the repetition of the "access-list 101" lines;
- there is no simple way of grouping hosts (e.g. hosts x,y,z are my web servers) or services (e.g. to allow imap traffic one actually needs to write 4 lines, to allow access to 4 sets of ports; many services use tcp and udp on the same ports, and one has to write 2 almost identical lines for each such service. This makes the lists long, hard to read and error prone[3];
- the only way to test what is being filtered is by inspection or to put your access-list in production and hope for the best; If a mistake is made, the network may stop functioning properly, or may be left in an insecure state.

With the above problems in mind, we would like to have a tool that solved the problems above, but we also would like it to use a similar notation, but it should be easier to understand and modify; it should also be easy to produce configurations for the various versions of Cisco IOS, and possibly for other routers.

Instead of writing such a tool from scratch, we decided to embed the configuration language into Haskell. This allowed a very rapid prototyping and definition of the auxiliary libraries for the various functionalities of the tool. This also allows the Haskell type system to catch many of the errors that would have to be programmed into a (simple) typechecker/semantics analyser.

But will we have to teach Haskell to the network administrator? He will certainly have to change Haskell code in order to program new configurations! As we will see, the resulting syntax is actually quite similar to the one he is used to write, and he only needs to see and change the files that configure the router and/or the services. From that on, simple scripts can be used to call a Haskell interpreter and hide all the implementation details from the network administrator.

[3]The access-list at DI-UFPE has about 50 rules!

3 The Network Administrator View

The network administrator is never really exposed to the underlying programming language, since he is exposed only to a configuration file (actually a Haskell module), where he specifies the access-lists with a syntax very similar to the one he is used to, but now with much more flexibility and customization.

The administrator can define names for some of his hosts and for the networks he manages, and use those in the access-lists:

```
recife = host "150.161.2.1"
caruaru = host "150.161.2.202"
webservers = hosts [recife, caruaru]
```

He can now proceed to define names for services that are not predefined in the tool, if necessary. Therefore when he looks at the access-list he will see only service names. He can even create aliases for the existing names. Here are some of the definitions we will be using below, that are predefined:

```
ssh = service [tcp] [22] "ssh"
imap = service [tcp,udp] [143,220] "imap"
```

Notice that now we associate the service name directly with the fact that it uses tcp, udp or both, and to the set of ports it uses.

How do access-lists look now?

```
access-list 101
  = [permit established Connections From anyhost To anyhost,
     permit www Connections From anyhost To webservers,
     permit imap Connections From anyhost To recife,
     permit ssh Connections From anyhost To caruaru,
     deny ip Connections From anyhost To anyhost ]
```

We noticed that the Cisco notation was itself difficult to understand due to the order in which the commands are laid in the command line (`permit/deny` *fromHost toHost service*, which we believe should be `permit/deny` *service fromHost toHost*). We therefore changed the order and put extra keywords[4] to improve legibility.

The resulting access-lists are shorter and more informative, with host names, all of them have service names, there is no need for repeating lines for a given service, or for sets of hosts with the same service. It is certainly easier to explain and to discuss security policies based on this access-list than with the one we presented before.

Another strong point about embedding the notation into Haskell is that it is easy to embed a more verbose syntax (as presented) or a simpler one, closer to the one used by the Cisco, if it should be preferred by the user. These are simple changes in the library, in the definitions for the `permit` and `deny` functions. The more verbose syntax we presented above, which is much more understandable, was preferred by our systems administrator.

After creating the access-lists and definitions inside the Haskell program, the user can invoke predefined programs that generate the Cisco access-lists (both old and new versions), or a program to test what services (or ports) are being blocked.

[4] The `From` and `To` keywords are implemented just as dummy datatype constructors, to work as placeholders.

We have, therefore, extended and improved the original syntax and functionality, by providing a tool that the network administrator can use to program access-lists for any version of the Cisco IOS, since we can "generate code" for both target versions. The tool is also very useful for evaluating and discussing security policies, and very easy to use.

4 Conclusions

We showed how a firewall configuration language was embedded into Haskell, and how the simplicity of this approach was used to have a very useful tool written very quickly. The resulting *embedded* language provided users a similar (but better) syntax with many advantages over the original environment.

We believe that this same approach can be used for other domains which have similar problems, in particular systems configuration languages with poor syntax, semantics and/or functionality. The approach we suggest presents a way of (virtually) extending the functionality of such languages and equipments, allowing configuration and testing to proceed without directly programming the equipment. Of course one still has to make tests on the real equipment, but the risk of misconfiguration would have been reduced.

The tool has been used successfully to program access-lists for the Computer Science Department at UFPE (DI-UFPE). For large sites, which provide services from many servers, or that manage many subnetworks with different access-lists, the extra functionality provided by this tool should prove to be very useful in reducing the effort of maintaining access-lists.

The fact that the syntax of the access-lists is entirely defined in Haskell has the advantage that many syntax and semantic errors are caught immediately by the Haskell interpreter. Without this tool, syntax errors are found only when someone is actually loading the access-list into the router. The fact that the syntax and type checking is handled by Haskell has the disadvantage that error messages are not easy to understand by the user. But the error messages given by the Cisco IOS when the access-list is wrong are even less informative.

References

[Ell97] C. Elliott. Modeling interactive 3d and multimedia animation with an embedded language. In *Proceedings of the first conference on Domai-Specific Languages*, October 1997.

[HMGW96] P. Hudak, T. Makucevich, S. Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 3(6):465–483, 1996.

[HPW+92] P. Hudak, S. Peyton Jones, P. Wadler, et al. Report on the functional programming language Haskell. `http://www.haskell.org`.

[Hud96] P. Hudak. Building domain-specific embedded languages. *Computing Surveys*, 4(28A), December 1996.

[MLC98] J. Matthews, J. Launchbury, and B. Cook. Microprocessor specification in Hawk. In *1998 International Conference on Computer Languages*, 1998.