Nettle: A Language for Configuring Routing Networks

Andreas Voellmy and Paul Hudak

Yale University,
Department of Computer Science
andreas.voellmy@yale.edu, paul.hudak@yale.edu

Abstract. *Interdomain routing* is the task of establishing connectivity among the independently administered networks (called *autonomous systems*) that constitute the Internet. The protocol used for this task is the *Border Gateway Protocol* (BGP) [1], which allows autonomous systems to independently define their own route preferences and route advertisement policies. By careful design of these BGP policies, autonomous systems can achieve a variety of objectives.

Currently available configuration and policy languages are low-level and provide only a few basic constructs for abstraction, thus preventing network operators from expressing their intentions naturally.

To alleviate this problem, we have designed *Nettle*, a domain-specific embedded language (DSEL) for configuring BGP networks, using Haskell [3] as the host language. The embedding in Haskell gives users comprehensive abstraction and calculation constructs, allowing them to clearly describe the ideas generating their BGP policies and router configurations. Furthermore, unlike previous router configuration and policy languages, Nettle allows users to both specify BGP policies at an abstract, network-wide level, and specify vendor-specific router details in a single uniform language.

We have built a compiler that translates Nettle programs into configuration scripts for XORP [4] routers and a simulator that allows operators to test their network configurations before deployment.

1 Introduction

Given the importance of the Internet, one would expect that it is well designed, that the principles upon which it operates are well understood, and that failures are due primarily to hardware crashes, network cable interruptions, and other mechanical or electrical failures. Unfortunately, and perhaps surprisingly, this is not the case.

The protocols that control the Internet were not designed for the way in which it is used today, leading to non-standard usage and ad hoc extensions. Some of the most fundamental principles upon which the Internet operates are poorly understood, and recent attempts to understand them better have revealed that in fact the protocols themselves permit network instabilities and outages

W.M. Taha (Ed.): DSL 2009, LNCS 5658, pp. 211–235, 2009.
© IFIP International Federation for Information Processing 2009

[5]. Furthermore, most of these problems are manifested not through hardware failures, but through software bugs, logical errors, or malicious intrusions.

The Internet has grown dramatically in size since its inception. The lack of centralized control is what has enabled this growth, and as a result the Internet is essentially owned and operated by thousands of different organizations. Most of these organizations (such as the many ISPs) share common goals – for example they all want messages to reach their destinations, and they do not want message content compromised. Because of this alignment of interests, the Internet, for the most part, works.

But this lack of centralized control is also a source of problems. For starters, the goals of the various entities that control the Internet are not always completely aligned. Different business models, economic incentives, engineering decisions, desired traffic patterns, and so on, can lead to conflicting interests in the control of the Internet. These conflicts in turn can lead to oscillations, deadlocks, and other kinds of instabilities in the forwarding of messages [5].

Furthermore, the standard protocols have become increasingly complex (mostly to accommodate the ever-increasing range of interests) and the effect of altering various parameters is not completely understood. Even detecting errors that lead to anomalies in Internet traffic is difficult, as they are often intermittent and difficult to localize.

To make matters worse, the scripting languages used to specify the behaviors of all major routers are all different, and are all astonishingly low level. Even if one knows exactly what behavior is desired, errors in configuring the network are quite common. The scripting languages have few useful constraints (such as a type system), no abstraction mechanisms, poorly understood semantics, and mediocre tools for development and testing. Indeed, it has been estimated that almost half of all network outages are a result of network misconfigurations [6].

1.1 A Language-Centric Solution

The goal of our research is to use modern programming language ideas to help ameliorate many of the aforementioned problems. We call our language and accompanying tools *Nettle*, and we have implemented it as a domain-specific embedded language [7] [8], using Haskell as a host. Nettle users express their router configurations in a high-level, declarative way, and the Nettle compiler generates the low-level scripting code for specific back-end routers (currently XORP). This allows network operators to specify goals without becoming mired in implementation details.

Specifically, Nettle offers the following advantages:

1. Nettle works with existing protocols and routing infrastructure; no changes to the routers themselves are needed.
2. Nettle takes a local-network-wide view, allowing operators to specify the entire local network routing configuration as one cohesive policy.
3. Nettle separates implementation- and hardware-specific concerns from logical routing concerns, resulting in a platform-neutral approach in which a single policy specification can be given for a heterogenous collection of routers.

4. Nettle permits multiple routing policies for the same network, and provides a flexible and safe way to merge their behaviors.
5. The embedding of Nettle in Haskell allows users to safely define their own configuration abstractions and invent new configuration methods.
6. By defining high-level policies in Nettle, it is possible to prevent global anomalies (such as route oscillations) if enough nodes on the Internet adopt the same strategy.

We have implemented a compiler for Nettle that generates configuration scripts for the XORP [4] open-source router platform. We have also implemented a BGP network simulator that allows Nettle users to explore the behavior of their BGP configurations under a variety of user-specified scenarios.

In the remainder of this paper we first give a brief introduction to network routing and BGP. We then introduce Nettle's salient features through a series of small examples and describe a complete network configuration. We then show how Nettle can be used to safely capture complex policy patterns as policy-generating functions. Finally, we discuss implementation issues and briefly compare Nettle with XORP and RPSL.

2 Introduction to Networks, Routing and BGP

A communication network is a distributed system that supports point-to-point messaging among its nodes. The behavior of such a network is typically decomposed into *forwarding* and *routing* processes. By *forwarding process* (*routing process*), we mean the collective forwarding (routing) behavior of all the nodes, rather than the behavior of a single process. Forwarding consists of sending messages between nodes, whereas routing consists of establishing paths between nodes. The relationship between forwarding and routing is that the forwarding process uses the paths established by the routing process. The routing process is dynamic – the paths between nodes may change over time. This is essential in networks in which communication links and nodes can fail. In such situations, the routing process typically responds to events by recomputing paths to avoid failed links or nodes.

In networks that are administered by a single organization, a routing process is typically used to compute least-cost paths among nodes, where cost is often a communication-link metric incorporating information such as bandwidth, reliability, distance or other qualities. On the other hand, networks such as the Internet, which consist of many independently administered sub-networks, also need to support point-to-point messaging. Such networks do not use a single routing process for a variety of reasons. One reason is that different local concerns may result in the use of different communication cost metrics. Another reason is that while networks need to establish some global connectivity, for security reasons they often wish to restrict visibility of their local internal structure.

Instead, such networks typically deploy their own *intra*-network routing process for their own nodes, and a single *inter*-network routing process to communicate with the rest of the network. The inter-network routing process is often

based on a high-level *policy* that is not necessarily based on a least-cost algorithm – it might also factor in economic incentives, traffic engineering, security concerns, and so on. Another key aspect of the inter-domain routing process is that it typically involves an exchange of messages that announce the availability of routes, and these announcements carry certain attributes that allow networks to make policy-based decisions about which routes to use.

The Internet community has established several standard protocols to implement routing processes. In Internet parlance, an intra-network routing process is called an *Interior Gateway Protocol* (*IGP*) and two commonly used IGP protocols are *Open Shortest Path First* (*OSPF*) and *Routing Information Protocol* (*RIP*). The independently administered networks on the Internet are called *Autonomous Systems* (*ASes*) and *domains* interchangeably. The interdomain routing protocol in use in the Internet is *Border Gateway Protocol* (*BGP*) [1]. Autonomous systems are assigned globally unique 32-bit *Autonomous System Numbers* (*ASNs*) by the *Internet Assigned Numbers Authority* (*IANA*). An *Internet Service Provider* (*ISP*) is an AS whose primary purpose is to provide Internet access to other ASes in exchange for payment.

2.1 BGP

BGP [1] is our main focus in this paper. BGP is essentially a routing process that is parameterized by a policy, just as a higher-order function is parameterized by a functional argument. In order to understand BGP policy, we need to understand some basic networking concepts.

An IP address is a 32 bit value, commonly written as four bytes (or *octets*, in Internet parlance), as in $a.b.c.d$. A subset of addresses, called an *address prefix*, is denoted by an address followed by a number between 0 and 32, as in $a.b.c.d/e$. A prefix $a.b.c.d/e$ corresponds to all addresses whose leftmost e bits match the leftmost e bits of $a.b.c.d$. More formally, we can write a prefix $a.b.c.d/e$ and address $w.x.y.z$ as sequences of bits $a_1a_2\dots a_e$ and $w_1w_2\dots w_{32}$; the address $w_1w_2\dots w_{32}$ is contained in prefix $a_1a_2\dots a_e$ if $a_i = w_i$ for all $1 \leqslant i \leqslant e$.

Nodes running BGP communicate by announcing routes to each other. These route announcements carry, among other data, the following information:

1. An address prefix, representing all addresses reachable using this route;
2. A sequence of AS numbers, called the *AS path*, which represents the ASes that messages will traverse when forwarded along this route;
3. *Community attributes*, 32-bit values that act as data used for ad-hoc communication between different ASes.

BGP allows networks to assign numeric *local preference* attributes to routes. These local preference values are then used as the primary criterion in the *BGP decision process*, which is used to choose the best route among several to the same prefix. The BGP decision process is a prefix-wise lexicographic ordering of routes based on their attributes, in the following order:

1. local preference
2. shortest AS path length

3. lowest origin type
4. lowest MED
5. eBGP-learned over iBGP-learned
6. lowest IGP cost to border router
7. lowest router ID (used as a tie-breaker)

The BGP decision process roughly implements shortest AS-path routing, while allowing networks to override this behavior by assigning non-default local preferences to routes. This allows networks to implement a variety of policies. For example, an operator may decide not to choose any paths that traverse a particular set of autonomous systems. Or they may prefer routes through peer A, except when peer B offers a route that also goes through peer C. Or they may want routes to some subset of destinations, such as universities, to go through one peer, while commercial traffic goes through another peer.

BGP routers usually also support *filter policies* and *export modifiers.* Filters are used to remove some received routes from consideration by the decision process or to prevent some routes from being announced to peers. Export modifiers are used to alter the attributes of routes that are announced to peers. For example, a network may add its AS number several times to the AS path of a route so that it appears longer.

We can organize these policies into two sorts, *usage policy* and *advertising policy.* Usage policy governs which routes are considered and how they are chosen for use by the network, while advertising policy governs which routes are offered to neighboring networks and with what attributes those routes are advertised. The usage policy determines how traffic flows out of an autonomous system, while advertising policy influences (but does not determine) how traffic will flow into the autonomous system. We will call a filter used to eliminate routes from consideration in the decision process *use filters* and filters used to prevent route announcements *ad filters.* If we represent the set of neighboring BGP routers as P, we can write the types of the policy functions as follows:

- $\mathit{useFilter} :: P \times \mathit{Route} \rightarrow \mathit{Bool}$
- $\mathit{preference} :: \mathit{Route} \rightarrow \mathbb{N}$
- $\mathit{adFilter} :: P \times \mathit{Route} \rightarrow \mathit{Bool}$
- $\mathit{adModifier} :: P \times \mathit{Route} \rightarrow \mathit{Route}$

Community Attributes. Community attributes are a crucial tool allowing networks to communicate about routes. Community attributes are 32-bit numeric values, commonly written as $a : b$, where a, b are 16-bit values. While there are a few community values that have an established meaning, such as the `NO_EXPORT` community value, most community values are available for ad hoc use between BGP neighbors. For example, an ISP may allow its customers to dynamically alter the preference levels of routes announced by the customers – these preferences are encoded as community attributes.

Internal BGP. The BGP protocol has two sub-protocols: IBGP (Internal BGP) and EBGP (External BGP). As the names suggest, EBGP governs the

interaction with external peers, while IBGP governs the interaction with internal BGP-speaking peers. IBGP's primary purpose is to ensure that all the BGP routers within an AS know the same routes and that they make decisions consistently.

2.2 Protocol Interaction

As mentioned earlier, an AS runs an IGP to establish routes to internal nodes and BGP to establish routes to the wider network. Routers running BGP typically run the IGP protocol also. At these routers the IGP and BGP protocols may interact, and this interaction is called called *route redistribution*. Typically, a BGP router will inject some routes into the IGP protocol so that these routes propogate to the non-BGP speaking routers in the network. To avoid overwhelming and potentially destabilizing the IGP, BGP routers are usually configured to advertise only a few aggregated routes, such as the default route 0.0.0.0/0.

3 Nettle Tutorial

In this section we introduce the salient features of the Nettle language, illustrating their use through small examples.

Preliminaries. Since Nettle is embedded in Haskell, it inherits the syntax of Haskell, and gives the user access to all of the power of Haskell. We assume that the reader is familiar with basic Haskell syntax.

Throughout this paper we have typeset some Nettle operators with more pleasing symbols. In particular, $\wedge$ is written as `/\` in Haskell, $\vee$ as `\/`, $\rhd$ as `>>>`, $/\!\!/$ as `//`, and $|||$ as `|||`.

An *IP address* such as 172.160.27.15 is written in Nettle as *address* 172 160 27 15. An address prefix such as 172.160.27.15/16 is written in Nettle as *address* 172 160 27 15 $/\!\!/$ 16, i.e. the $/\!\!/$ operator takes an address and a length and returns a prefix.

Network operators usually write community attributes as $x : y$, where x is the higher-order 16 bits and y is the lower-order 16 bits of the data field. By convention, x is used to encode the local AS number. In Nettle we represent a community attribute as $x ::: y$.

3.1 Routing Networks

A routing network consists of a collection of routers participating in one or more routing protocols. For example, typical networks run BGP on border routers; some internal routing protocol, such as OSFF or RIP, on all routers; and additionally configure some routes statically, that is, not learned through a dynamic routing protocol. Currently Nettle supports only two routing protocols, *BGP* and *Static*.

At the highest level, a Nettle program describing a BGP policy has the form:

$$nettleProg = routingNetwork\ bgpNet\ staticNet\ redistPolicy$$

where

1. *bgpNet* is a description of the BGP network.
2. *staticNet* is a description of the static protocol.
3. *redistPolicy* describes how the BGP and static protocols interact.

bgpNet is typically defined as follows:

$$bgpNet = bgpNetwork\ asNum\ bgpConns\ prefs\ usefilter\ adfilter\ admodifier$$

where:

1. *asNum* is the AS number of the AS whose routing is being defined.
2. *bgpConns* is a list of connections.
3. *prefs* is an assignment of preference levels, i.e. integers, to routes and is used to determine which routes to use.
4. Given a set of routes received by a router, *usefilter* discards ones it doesn't want.
5. Given a set of routes known by a router, *adfilter* discards those that it does not wish to advertise to neighbors.
6. *admodifier* is a function which may change attributes of a route as it is exported; this used to influence neighbors routing decisions and to ultimately influence incoming traffic.

In the following subsections we will see how each of these constituent pieces is generated.

Routers. Several aspects of routing policy, such as static route announcements and route redistribution, are specified in terms of particular routers in the network. Additionally, the Nettle compiler will need router-specific details, such as hardware configurations, in order to compile a Nettle program. Nettle programs therefore need to know the set of routers to be configured.

In order to maintain modularity, Nettle policies are written in terms of an abstract router type that router-specific data types implement. For example, we can declare an abstract router *r1* , implemented by a XORP router, with the following code:

```
r1 = router r1xorp
where r1xorp           = xorpRouter xorpBgpId xorpInterfaces
      xorpInterfaces = [ifaceEth0]
      ifaceEth0        = xorpInterface "eth0" "data" [vifEth0Eth0]
      vifEth0Eth0      = let block       = address 200 200 200 2 // 30
                             bcastAddr = address 200 200 200 3
                         in vif "eth0" (vifAddrs (vifAddr block bcastAddr))
```

Here the *router* function hides the specific type of router. Policy is then written in terms of the abstract router type, as will be shown in the following sections. This design separates router-specific details from abstract policy and allows configurations to be written modularly. In particular, it allows users to change the router platform of a router without changing any routing policy. Furthermore, this design allows Nettle to support configurations of a heterogeneous collection of router types within a single language.

In the following examples, we assume that we have defined routers *r1*, *r2*, and *r3*.

Static Routing. Static routes are specified simply by describing the address prefix, the next-hop router, and the router that knows of the route. For example, the following describes a static routing configuration with three routes, known at two routers:

```
staticConfig [
  staticRoute r1 (address 172 160 0 0 // 16) (address 63 50 128 1),
  staticRoute r1 (address 218 115 0 0 // 16) (address 63 50 128 2),
  staticRoute r2 (address 172 160 0 0 // 16) (address 63 50 128 1)]
```

We note that a standard Haskell **let** expression can be used to rewrite this as:

```
let ip1 = address 172 160 0 0 // 16
    ip2 = address 218 115 0 0 // 16
    ip n = address 63 50 128 n
in staticConfig [staticRoute r1 ip1 (ip 1),
                 staticRoute r1 ip2 (ip 2),
                 staticRoute r2 ip1 (ip 1)]
```

which is arguably easier to read. Even this simple kind of abstraction is not available in most (if any) router scripting languages.

BGP Connections. A connection specification consists of a set of *BGPConnection* values. For example, the following two connections describe external and internal connections, respectively:

```
conn1 = externalConn r1 (address 100 100 1 0) (address 100 100 1 1) 3400
conn2 = internalConn r1 (address 130 0 1 4) r3 (address 130 0 1 6)
```

The first line declares that router *r1* has a BGP connection with an external router from AS 3400 and that the address for *r1* on this connection is 100.100.1.0 and the peer's address is 100.100.1.1. The second line declares that *r1* and *r3* are connected via IBGP using addresses 130.0.1.4 for *r1* and 130.0.1.6 *r3*.

Subsets of Routes. At the core of Nettle lies a small language of route predicates for specifying sets of routes. This language is used to apply different policies to different sets of routes. The language, which is designed with an eye towards implementation on available router platforms, consists of a set of atomic

predicates; a conjunction operator, $\wedge$, denoting the intersection of two subsets of routes; and a disjunction operator, $\vee$, denoting the union of two subsets of routes. For example,

$$\mathit{nextHopEq}\ (\mathit{address}\ 128\ 32\ 60\ 1) \vee \mathit{taggedWith}\ (5000 ::: 120)$$

denotes the set of routes whose next hop routers have address 128.32.60.1 or those which are tagged with community attribute 5000:120.

Another example is:

$$\begin{aligned}&\mathit{destInSet}\ [\mathit{address}\ 128\ 32\ 60\ 0\ /\!/\ 24, \mathit{address}\ 63\ 100\ 0\ 0\ /\!/\ 16]\\ &\wedge \mathit{taggedWithAnyOf}\ [5000 ::: 120, 7500 ::: 101]\end{aligned}$$

which denotes the set of routes for destination prefixes 128.32.60.0/24 or 63.100.0.0/16 and which are marked with one or more of community attributes 5000:120 or 7500:101.

The *asSeqIn* predicate allows users to specify complex conditions on BGP routes, as in the following example which requires that routes originate at AS 7000, pass through any number of networks, and finally pass through either AS 3370 or 4010 once followed by AS 6500:[1]

$$\begin{aligned}\mathit{asSeqIn}\ (&\mathit{repeat}\ (i\ 7000) &&\triangleright\ \mathit{repeat}\ \mathit{any}\ \triangleright\\ &(i\ 3370\ |||\ i\ 4010) &&\triangleright\ \mathit{repeat}\ (i\ 6500))\end{aligned}$$

The full collection of predicates is shown in Figure 1.

Using Haskell's standard abstraction mechanisms, this predicate language allows users to write new predicates. For example, we can define a predicate that matches a path exactly, while allowing for prepending, as follows:

$$\begin{aligned}&\mathit{pathIs} && :: [\mathit{ASNumber}] \to \mathit{RoutePredicate}\ \mathit{BGPT}\\ &\mathit{pathIs}\ \mathit{xs} && = \mathit{asSeqIn}\ \$\ \mathit{foldr}\ (\lambda a\ r \to \mathit{repeat}\ (i\ a)\ \triangleright\ r)\ \mathit{empty}\ \mathit{xs}\end{aligned}$$

It is impossible to express this kind of predicate in any router scripting language that we are aware of.

Usage and Advertising Filters. Filters play a role in both usage and advertising policy; a usage filter prevents some routes from being considered for use, while an advertising filter prevents some routes from being advertised to peers. Nettle allows users to declare filters based on predicates using the *reject* function. For example,

$$\mathit{reject}\ (\mathit{destEq}\ (\mathit{address}\ 128\ 32\ 0\ 0\ /\!/\ 16))$$

is a filter that rejects routes to destinations 128.32.0.0/16.

Nettle also allows users to specify connection-dependent filters, i.e. maps associating BGP connections with usage (or advertising) filters. For example, the

[1] In this paper *repeat* is the kleene star operation on regular expressions, not the *repeat* function in Haskell's *Prelude* module.

$$
\begin{array}{lllll}
\mathit{destEq} & :: \mathit{Protocol}\ p \Rightarrow & \mathit{AddressPrefix} & \rightarrow \mathit{RoutePred}\ p \\
\mathit{destNotEq} & :: \mathit{Protocol}\ p \Rightarrow & \mathit{AddressPrefix} & \rightarrow \mathit{RoutePred}\ p \\
\mathit{destInRange} & :: \mathit{Protocol}\ p \Rightarrow & \mathit{AddressPrefix} & \rightarrow \mathit{RoutePred}\ p \\
\mathit{destInSet} & :: \mathit{Protocol}\ p \Rightarrow & [\mathit{AddressPrefix}] & \rightarrow \mathit{RoutePred}\ p \\
\mathit{nextHopEq} & :: & \mathit{Address} & \rightarrow \mathit{RoutePred}\ \mathit{BGPT} \\
\mathit{nextHopInRange} & :: & \mathit{Address} \rightarrow \mathit{Address} & \rightarrow \mathit{RoutePred}\ \mathit{BGPT} \\
\mathit{nextHopInSet} & :: & [\mathit{Address}] & \rightarrow \mathit{RoutePred}\ \mathit{BGPT} \\
\mathit{asSeqIn} & :: & \mathit{RegExp}\ \mathit{ASNumber} & \rightarrow \mathit{RoutePred}\ \mathit{BGPT} \\
\mathit{taggedWith} & :: & \mathit{Community} & \rightarrow \mathit{RoutePred}\ \mathit{BGPT} \\
\mathit{taggedWithAnyOf} & :: & [\mathit{Community}] & \rightarrow \mathit{RoutePred}\ \mathit{BGPT} \\
\mathit{all} & :: \mathit{Protocol}\ p \Rightarrow & \mathit{RoutePred}\ p & \\
\mathit{none} & :: \mathit{Protocol}\ p \Rightarrow & \mathit{RoutePred}\ p & \\
(\wedge) & :: \mathit{Protocol}\ p \Rightarrow & \mathit{RoutePred}\ p \rightarrow \mathit{RoutePred}\ p \rightarrow \mathit{RoutePred}\ p & \\
(\vee) & :: \mathit{Protocol}\ p \Rightarrow & \mathit{RoutePred}\ p \rightarrow \mathit{RoutePred}\ p \rightarrow \mathit{RoutePred}\ p &
\end{array}
$$

Fig. 1. The language of route predicates

following usage filter rejects routes learned over connection *c1* which either (1) are for block 128.32.0.0/16 and are tagged with community 5000:120, or (2) are tagged with community 12345:100, while for other connections it rejects only routes that pass through network 7000:

$$
\begin{array}{l}
\mathit{usefilter}\ c = \\
\quad \mathbf{if}\ c \equiv \mathit{c1} \\
\quad \mathbf{then}\ \mathit{reject}\ ((\mathit{destEq}\ (\mathit{address}\ 128\ 32\ 0\ 0\ /\!\!/\ 16) \\
\qquad\qquad\qquad \wedge\ \mathit{taggedWith}\ (5000 ::: 120)) \vee \mathit{taggedWith}\ (12345 ::: 100)) \\
\quad \mathbf{else}\ \mathit{reject}\ (\mathit{asSeqIn}\ (\mathit{repeat}\ \mathit{any}\ \rhd\ \mathit{repeat}\ (i\ 7000)\ \rhd\ \mathit{repeat}\ \mathit{any}))
\end{array}
$$

Route Preferences. The central part of BGP usage policy consists of route preferences. In Nettle, users specify route preferences by giving a numerical rank, with higher being more preferred, to sets of routes, which are specified using route predicates. To do this we can use the *route conditional* expressions *cond* and *always*. For example, the expression[2]

$$
\begin{array}{l}
\mathit{cond}\ (\mathit{taggedWith}\ (5000 ::: 120))\ 120 \\
\quad \$\ \mathit{cond}\ (\mathit{taggedWith}\ (5000 ::: 100))\ 100 \\
\quad \$\ \mathit{cond}\ (\mathit{taggedWith}\ (5000 ::: 80))\ 80 \\
\quad \$\ \mathit{always}\ 100
\end{array}
$$

ranks routes with community 5000:120 at rank 120, routes not tagged with 5000:120 but tagged with 5000:100 at rank 100, and so on, until it finally matches all remaining routes with rank 100.

[2] Note that $f \$ x = f\ x$. Using this function allows us to avoid writing some parantheses; for example $g\ \$\ f\ \$\ x = g\ (f\ x)$.

Route Modifiers and Guarded Route Modifiers. The central part of BGP advertising policy consists in modifying route attributes as routes are advertised to peers, so as to communicate intentions about routes, or influence how others prefer routes. Intentions about routes can be communicated through the use of community attributes, while AS path prepending can influence peers' decisions by increasing the apparent path length of the route. Nettle provides a small language for expressing route modifiers. For example, the modifier:

$$\textit{tag}\ (5000 ::: 130)$$

denotes a function mapping a BGP route to the same BGP route tagged with community 5000:130, while the modifier:

$$\textit{prepend}\ 65000$$

represents a function mapping a BGP route to the same BGP route with the AS number 65000 prepended to the AS path attribute. Two route modifiers f and g can be combined as $f \ \triangleright\ g$, which denotes the reverse composition of the modifiers denoted by f and g. For example,

$$\textit{tag}\ (5000 ::: 130)\ \triangleright\ \textit{prepend}\ 65000$$

represents the function which associates a BGP route with the same route tagged with community 5000:130 and with AS path prepended with AS number 65000. Finally, route modifiers *ident* and *unTag c* leave a route unchanged and remove a community value c from a route, respectively.

As with route predicates, route modifiers allow us to define new modifiers and use them wherever route modifiers are required. For example, we can define a modifier which prepends an AS number a specified number of times:

$$\textit{prepends}\ n\ \textit{asn} = \textit{foldr}\ (\ \triangleright\ )\ \textit{ident}\ \$\ \textit{replicate}\ n\ (\textit{prepend}\ \textit{asn})$$

Route modifiers can be combined with conditional statements to describe *guarded route modifiers*. For example,

$$\textit{cond}\ (\textit{taggedWith}\ (5000 ::: 120))\ (\textit{prepend}\ 65000)\ (\textit{always}\ (\textit{tag}\ 1000 ::: 99))$$

is a guarded route modifier that prepends the AS number 65000 to a route only if that route is tagged with community value 5000:120, and otherwise tags the route with community 1000:99.

Nettle allows route modifiers to be specified per connection and represents these as Haskell functions. For example,

$$\begin{aligned} \textit{adMod}\ c\ &|\ c \equiv \textit{c1} &&= \textit{always}\ (\textit{prepend}\ 65000) \\ &|\ c \equiv \textit{c2} &&= \textit{always}\ (\textit{prepends}\ 2\ 65000\ \triangleright\ \textit{tag}\ (65000 ::: 120)) \\ &|\ \textit{otherwise} &&= \textit{always}\ \textit{ident} \end{aligned}$$

Connecting Protocols. Nettle currently provides a simple mechanism for connecting protocols, a process also known as *route redistribution*, in which a subset of routes from one protocol are introduced into another protocol at a particular router. For example,

$$\textit{redistAt r1}\ (\textit{destEq}\ (\textit{address}\ 80\ 10\ 32\ 0\ /\!\!/\ 24))\ \textit{StaticProtocol BGPProtocol}$$

indicates that routes with destination 80.10.32.0/24 from the static protocol will be redistributed into the BGP protocol at router *r1*. Multiple redistribution statements can be combined with *redists*:

```
let p1 = redistAt r1 (destEq (address 80 10 32 0 // 24))
                     StaticProtocol BGPProtocol
    p2 = redistAt r2 (destEq (address 100 10 20 0 // 24))
                     StaticProtocol BGPProtocol
in  redists [p1, p2]
```

3.2 Compiling to Routers

By combining routing policy with information about the specific routers in the network, we can compile appropriate configuration files for each router. We can compile a configuration for a router in our network with the command *compile*, which returns a *String* value, as in the following example, which gives the configuration file for *r1*:

compile rNet r1

Although Nettle currently only provides a compiler for XORP routers, we ultimately intend the Nettle compiler to support all major router platforms.

3.3 Simulation

The Nettle library also provides a simple BGP simulator, whose implementation is based on the formal BGP model of Griffin et al [9]. The simulator provides a coarse approximation of BGP. In particular, it ignores timing details, and rather calculates routing tables for each BGP node in a network in rounds. In each round BGP nodes apply export policy to their best routes, exporting to neighboring nodes, and then apply usage policy to their newly received and currently known routes.

A significant limitation of the simulator is that it does not model IBGP and that it only models one connection per pair of peers. Therefore in order to simulate our Nettle BGP networks with this simulator, we need to approximate our network with a single BGP node, preserving the connections and policies of the original Nettle network as much as possible. Due to these limitations, the simulator is not yet useful for simulating the behavior of most large networks. However, for small networks which have only one connection per BGP router

and external peer, the simulator provides a useful tool for exploring the effects of policies, as the example in Section 4 shows.

We briefly describe how to use the simulator here. To create a simulation

bgpSimulation connGraph policyMap asNumberMap

where *connGraph*, of type *Graph* from the *Data.Graph* module, is the connectivity graph of the nodes in the network, where our network is a single node and nodes are identified with integers, and *policyMap* :: *Int* → *SimPolicy* and *asNumberMap* :: *Int* → *ASNumber* are maps giving for each node identifier the node's policy and its AS number respectively. The library provides a function *networkToSimPolicy* which calculates the simulator policy of the simulator node representing the home network, given a routing network and a map associating to each external node the BGP connection (part of the Nettle specification) connecting it to the home network. A simulation can then be run and viewed, using the *printSimulation* function, whose use is shown in Section 4.

4 Extended Nettle Example

We can now demonstrate a complete configuration in Nettle. The example is taken from Zhang et al [10], a Cisco text book on BGP configuration. The example is for the configuration of a network with AS number 300, which is "multi-homed", (i.e. it connects to the Internet via multiple providers), to providers networks AS 100 and AS 200. AS 300 has 3 BGP-speaking routers, *R1*, *R2*, and *R3* all of which have IBGP sessions among them. Router R1 is connected to AS 100 via a high-bandwidth OC-3 (Optical Carrier, 155 Mbit/s) connection, while R2 and R3 are links to AS 100 and AS 200 respectively over lower bandwidth DS-3 (Digital signal, 45 Mbit/s) links. The topology is shown in Figure 2. For the remainder of this section, we describe the policy from the point of view of the multi-homed network.

Our intention is that most traffic should flow over the OC-3 link to AS 100, while traffic to destinations that are customers of AS 200 should flow over the DS-3 link to AS 200. This policy will achieve some load balancing while also favoring short routes to customers of AS 200. We also plan for some failure scenarios: if the OC-3 link should fail, we would like to roughly balance traffic over the two DS-3 links; if the DS-3 to AS 200 fails, all traffic should flow over the OC-3, whereas if the DS-3 link to AS 100 fails, traffic should be unaffected.

In order to achieve these outbound traffic goals, we first request default and *partial* routes from both providers, meaning that our providers will advertise a default route to the Internet as well as more specific routes to their customers.[3] We can then implement our preference policy with two preference levels, which we will call *high* and *low*. We assign routes traversing the OC-3 link the *high* preference, and all other routes the *low* preference.

[3] This request is not part of BGP; this request is made by informal communication among the networks' administrators.

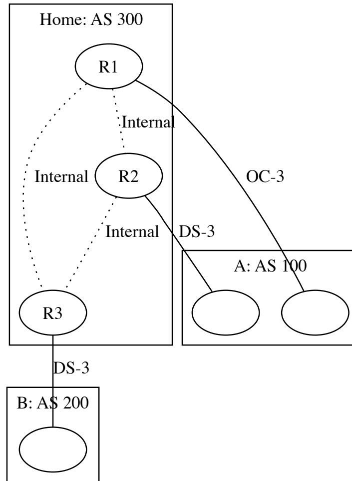

Fig. 2. The BGP connections for example 1

With this policy, we will never use a route traversing the DS-3 to AS 100 when the OC-3 is working, because 1) we assume that any route advertised by AS 100 will be advertised on both of its links to us and 2) we give a higher preference to routes over the OC-3 link. We will also never use a route traversing the DS-3 link to AS 200 when a route to the same prefix is available over AS 100. We will only use a route traversing the DS-3 to AS 200 when the prefix of that route is not announced by AS 100; this will only be the case when the prefix is for a customer of AS 200 that is also not a customer of AS 100.

We are more limited in our ability to influence inbound traffic patterns. To do this we must manipulate how our routes are advertised in order to influence the decisions that AS 100 and AS 200 make. To influence AS 100's traffic to the OC-3 link, we can prepend our AS number onto routes we advertise across the DS-3 link to AS 100. Assuming that AS 100 does not have explicit preferences set for routes, this will cause AS 100 to prefer the apparently shorter path to us over the OC-3. In order to discourage peers of AS 200 from choosing the route to us through AS 200, we prepend our AS number twice on the link to AS 200. However, in the situation Zhang et al [10] suppose, this amount of prepending causes AS 200 itself (as opposed to its neighbors) to choose the route to us through AS 100 and so the local traffic from AS 200 does not flow over the link with AS 200. To remedy this situation, we take advantage of policies which are known[4] be in place at AS 200. AS 200 has configured policies that assigns routes having community values 200:80, 200:100, and 200:120 the preferences 80, 100, and 120 respectively. By advertising routes to AS 200 with community attribute 200:120, we can cause AS 200 to prefer the direct route to us, while

[4] Knowledge of such policies is communicated in an ad-hoc fashion by neighboring network administrators.

```
import Nettle.Network
home = 300; asA = 100; asB = 200
conn12      = internalConn r1 (address 130 0 1 4) r2 (address 130 0 2 5)
conn13      = internalConn r1 (address 130 0 1 4) r3 (address 130 0 3 4)
conn23      = internalConn r2 (address 130 0 2 6) r3 (address 130 0 3 5)
conn_A_OC3 = externalConn r1 (address 100 100 100 0)
                              (address 100 100 100 1) asA
conn_A_DS3 = externalConn r2 (address 100 100 150 0)
                              (address 100 100 150 1) asA
conn_B_DS3 = externalConn r3 (address 200 200 200 0)
                              (address 200 200 200 1) asB
conns = [conn12, conn13, conn23, conn_A_OC3, conn_A_DS3, conn_B_DS3]
staticNet = staticConfig [route r1, route r2, route r3]
  where route r = staticRoute r (address 172 160 0 0 // 16)
                                (localAddr conn_B_DS3)
redistPolicy = redist [nofilter r | r ← [r1, r2, r3]]
  where nofilter r = redistAt r all StaticProtocol BGPProtocol
martians = [address 0     0   0 0 // 8,  address 10  0   0 0 // 8,
            address 172   0   0 0 // 12, address 192 168 0 0 // 16,
            address 127   0   0 0 // 8,  address 169 254 0 0 // 16,
            address 192   0   2 0 // 24, address 224 0   0 0 // 3,
            address 172   160 0 0 // 16]
adMod c | c ≡ conn_A_DS3 = cond all (prepend home) (always ident)
        | c ≡ conn_B_DS3 = cond all
                               (tag (200 ::: 120) ▷ prepends 2 home)
                               (always ident)
        | otherwise      = always ident
adFilter = const $ reject $ destNotEq homeBlock
  where homeBlock = address 172 160 0 0 // 16
net = routingNetwork bgpNet staticNet redistPolicy
  where bgpNet    = bgpNetwork home conns prefs usefilter adFilter adMod
        prefs     = cond (nextHopEq (peerAddr conn_A_OC3)) high
                         (always low)
        usefilter = const $ reject $ destInSet martians
        high      = 120
        low       = 100
```

Fig. 3. The Nettle configuration for the Zhang et al [10] example; the definitions of the router details *r1*,*r2*,*r3* are omitted

still prepending our AS number twice, thereby making the route appear long for peers of AS 200.

Finally, we declare use and ad filters. The use filter is a defensive measure which prevents us from accepting routes which are known to be incorrect. The ad filter ensures that we only advertise our own address block and do not inadvertently enable our providers to transit traffic through our network.

The Nettle code for this example, omitting the router-specific details, is shown in Figure 3.

We can use the simulator to check our understanding of the policy. We set up a simulation in which AS 100 announces only the default route, and AS 200 announces the default route and two more specific routes: one for destination 20.20.20.0/24 and another to destination 10.0.0.0/7. We expect that the 10.0.0.0/7 route will be filtered by our martian filter. In addition, we expect the default route over the OC-3 link to be preferred. We also check that we don't inadvertently provide transit traffic to our providers by exporting one provider's routes to the other. The simulation output, shown in Figure 4 confirms our understanding.

```
printSimulation sim
(Node = 1, AS = 300):
(    0.0.0.0 //  0, 100.100.100.1, [100],     120, [])
( 20.20.20.0 // 24, 200.200.200.1, [200,7763],100, [])
(172.160.0.0 // 16, 200.200.200.0, [],          0, [])

(Node = 2, AS = 100):
(172.160.0.0 // 16, 100.100.100.0, [300],0, [])
(    0.0.0.0 //  0, 100.150.150.88,[],   0, [])

(Node = 3, AS = 100):
(172.160.0.0 // 16, 100.100.150.0 ,[300,300],0, [])
(    0.0.0.0 //  0, 100.100.150.23,[],       0, [])

(Node = 4, AS = 200):
(172.160.0.0 // 16, 200.200.200.0 ,[300,300,300],0, [Community 200 120])
(    0.0.0.0 //  0,  10.23.140.223,[],           0, [])
( 20.20.20.0 // 24,  10.23.140.223,[7763],       0, [])
(   10.0.0.0 //  7,  10.23.140.223,[7763],       0, [])
```

Fig. 4. A sample simulation for the policy of Section 4

5 Nettle as a Basis for User-Defined Policy Languages

The most compelling advantage of Nettle, we believe, is its ability to serve as the basis for the development of higher-level and more specific configuration schemes or patterns. In this section we demonstrate two such schemes that can be precisely expressed in Nettle. These examples illustrate how Nettle enables policy to be written abstractly and at a high-level. Though it is possible to write policy-generating scripts in other languages, the Nettle language offers safety guarantees: if the policy-generating functions type check, then we are sure that the policies they generate will be free from many simple errors. This simple, but substantial verification may allow authors of such functions to write policy-generators more quickly and be more confident in the correctness of the resulting policy.

5.1 Hierarchical BGP in Nettle

Gao and Rexford [11] found that commercial relationships between networks on the Internet are roughly of two types: customer-provider relationships and peer relationships. In a customer-provider relationship, one network (the customer) pays the other (the provider) to connect them to the rest of the Internet. In a peer relationship, two networks agree to allow traffic to and from customers of the networks to pass directly between the networks, without payment. These relationships strongly constrain BGP policy in two ways. On the one hand, providers are obliged to support traffic to and from customers. On the other hand networks have an incentive to use customer-learned routes, since using such routes incurs no cost and since customers may pay by amount of data transmitted. These constraints, as described in Griffin et al. [9] can be summarized as follows:

1. Customer routes must be preferred to peer routes, and customer and peer routes must be preferred over provider routes.
2. Customer routes are be advertised to all neighbors, whereas peer and provider routes can only be shared with customers.

Hierarchical BGP (HBGP) is an idealized model of BGP in which networks relate in the two ways described above and follow the guidelines described. Griffin et al. [9] have shown that these guidelines, if followed by all networks on the Internet, would guarantee the abscence of many routing anomalies, including global routing oscillations.

We can express both of these guidelines in Haskell functions that generate Nettle policy given the classification of peerings into HBGP relationship types, and indeed, we have implemented such functions in the *Nettle.HBGP* module. We will need to generate both preferences and advertising filters to implement these guidelines, and we do so with the following two functions:

$$
\begin{aligned}
&\mathit{hbgpAdFilter} :: [(\mathit{ASNumber}, \mathit{PeerType})] \rightarrow \mathit{BGPConnection}\ r \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow \mathit{Filter}\ \mathit{BGPT} \\
&\mathit{hbgpPrefs} \quad\ :: [(\mathit{ASNumber}, \mathit{PeerType})] \rightarrow \mathit{PartialOrder}\ \mathit{ASNumber} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow \mathit{Cond}\ \mathit{BGPT}\ \mathit{Preference}
\end{aligned}
$$

where we use the following enumerated type to stand for the HBGP relationship types:

$$\textbf{data}\ \mathit{PeerType} = \mathit{Customer} \mid \mathit{Peer} \mid \mathit{Provider}$$

The HBGP preference guidelines do not constrain preferences among neighbors of a given relationship type and we take advantage of this by having the *hbgpPreference* function also take a *PartialOrder ASNumber* argument, which represents route preferences among the neighbors. The function will attempt to satisfy both the HBGP preference constraint and the given partial order, and will report an error if these are incompatible. We omit the implementation of these functions here, but we show how these can be used to easily create a Nettle configuration that follows HBGP guidelines.

In this example, we show only the preference and advertising filter components of the policy for a network with several customers and peers, and one provider. First, we we define the set of external network numbers and a map to classify external neighbors into HBGP peer types:

$$\begin{aligned}
&home = 100; cust1 = 200; cust2 = 300; cust3 = 400 \\
&cust4 = 500; peer1 = 600; peer2 = 700; prov = 800 \\
&pTypes = [(cust1, Customer), (cust2, Customer), (cust3, Customer), \\
&\qquad (cust4, Customer), (peer1, Peer), (peer2, Peer), (prov, Provider)]
\end{aligned}$$

We can then easily define our preferences to be HBGP compatible preferences, where we also give our preference of networks:

$$\begin{aligned}
&prefs = hbgpPrefs\ pTypes\ basicPrefs \\
&\quad \textbf{where}\ basicPrefs = [(cust1, cust2), (cust1, cust4), (cust3, cust4), \\
&\qquad\qquad (peer2, peer1), (prov, prov)]
\end{aligned}$$

We also add the HBGP advertising filters to our policy:

$$adFilter = hbgpAdFilter\ peerTyping$$

Compiling our example for a single Xorp router gives a configuration file roughly 400 lines long, much of it consisting of preference setting commands, such as:

```
term impterm13 {
 from {
  as-path: "^200|(200 [0-9][0-9]*( [0-9][0-9]*)*)$"
 }
 then {
  localpref: 105
 }
}
term impterm14 {
 from {
  as-path: "^300|(300 [0-9][0-9]*( [0-9][0-9]*)*)$"
 }
 then {
  localpref: 104
 }
}
...
```

as well as export filters implementing the HBGP scope preferences, such as:

```
term expterm7 {
 to {
  neighbor: 130.6.1.1
  as-path: "^700|(700 [0-9][0-9]*( [0-9][0-9]*)*)$"
 }
```

```
  then {
   reject
  }
 }
 term expterm8 {
  to {
   neighbor: 130.6.1.1
   as-path: "^800|(800 [0-9][0-9]*( [0-9][0-9]*)*)$"
  }
  then {
   reject
  }
 }
```

This example illustrates the utility of Nettle in capturing configuration patterns and thus enabling network configurations to be specified more succinctly and at a higher level of abstraction.

5.2 Dynamically Adjustable Customer Policy

Providers often receive requests from customers to modify their policy in order to achieve some customer objective. Furthermore, the customer may alter their objective at a later date, and request another change in provider policy. In order to avoid having to repeatedly update their configurations, and thereby increase the risks of introducing accidental errors, providers often provide a way for customers to dynamically alter provider policy. This is typically done through the use of the community attribute. Providers typically configure policies to match on particular community values, which when matched perform some operation, such as setting the preference level of the route to a particular value, or influencing the advertisement of the route. The network configured in Section 4 made use of such a policy when announcing routes to AS 200 by tagging these routes with an appropriate community value.

Zhang et al. [10] describe four commonly occurring types of dynamically adjustable policies used by ISP's. These policy types allow customers to do the following:

1. Adjust preference levels of announced routes.
2. Suppress advertisement of routes to peers by peer type.
3. Suppress advertisement of routes to peers by peer AS number.
4. Prepend ISP AS Number to routes announced to peers with a certain AS Number.

To allow a customer to adjust preference levels, an ISP with AS Number 1000 could add policy to match routes having community attributes 1000:80, 1000:90, ... 1000:120 and set the local preference to 80, 90,..., 120, respectively. To suppress by peer type, AS 1000 could add filters matching community values 1000:210, 1000:220, and 1000:230 where these filters cause matching routes not to be advertised to providers, peers, or customers, respectively. To suppress by AS number,

AS 1000 could add a filter for every neighboring network, with AS number A, matching community 65000:A and suppressing advertisement to A upon matching. To achieve dynamically adjustable prepending, AS 1000 would add a filters for every neighboring network, with AS number A, of the following form: match comunity 65x00:A, where x is a digit from 0-9, and upon matching, prepend 1000 to the AS path x times.

It is easy to see that such policy will dramatically increase the size of the router configurations with many simple definitions and that writing these by hand will substantially increase the likelihood of introducing errors into configurations. A better solution is to capture these configuration patterns as functions producing Nettle expressions, and instantiating these patterns in configurations by calling these functions. Indeed, we have done exactly this in the module *Nettle.DynamicCustomerPolicy*.

We define a function *adjustablePrefs* which returns customer-adjustable Nettle preferences, given a list of preference levels which may be set. An example is

$$\textit{adjustablePrefs homeASNum custASNum } (\textit{always } 100)\ [80, 100, 120]$$

which denotes preference policy which allows customer number *custNum* to adjust the preference level of their routes among 80, 100, and 120 by tagging their routes with communities *homeASNum*:::80, *homeASNum*:::100, and *homeASNum*:::120 respectively, and otherwise defaults to preference level 100. The implementation of *adjustablePrefs* is relatively straightforward, so we show it here:

$$\begin{aligned}
&\textit{adjustablePrefs cust home prefElse prefLevels} = \\
&\quad \textit{foldr f prefElse prefLevels} \\
&\quad \textbf{where } f\ p\ e = \textit{cond}\ (\textit{routeFrom cust} \wedge \textit{taggedWith}\ (\textit{home} ::: p))\ p\ e \\
&\quad\qquad\quad \textit{routeFrom asn} = \textit{asSeqIn}\ (\textit{repeat}\ (i\ \textit{asn})\ \triangleright\ \textit{repeat any})
\end{aligned}$$

The *routeFrom cust* predicate ensures that only the specified customer can adjust their own routes.

We can accomplish dynamic prepending with the *adjustablePrepending* function, and an example of its use is the following:

$$\begin{aligned}
&\textit{adjustablePrepending homeNum custNum}\ [(91, 1), (92, 2), (93, 3), (94, 4)] \\
&\qquad\qquad\qquad\qquad [44000, 13092, 6231]\ (\textit{always ident})
\end{aligned}$$

which denotes connection-dependent advertising modifier policy which allows customer *custNum* to adjust the prepending done by the provider network when advertising to networks 44000, 13092, and 6231 such that communities 91:44000, 92:44000, ..., 94:44000, 91:13092, ... , 94:6231 correspond to prepending 1, 2, 3, and 4 times, to the specified network, respectively. The implementation of *adjustablePrepending* is more involved than for *adjustablePrefs*, but is nonetheless not too complicated:

$$\begin{aligned}
&\textit{adjustablePrepending home cust octetsAndTimes nets gmodElse conn} = \\
&\quad \textbf{if} \quad \textit{peerASNum conn}\ `\textit{member}`\ \textit{nets}
\end{aligned}$$

```
then foldr f gmodElse octetsAndTimes
else gmodElse
where f (o, t) e = cond (routeFrom cust
                          ∧ taggedWith (o ::: (peerASNum conn)))
                        (prepends t home) e
```

6 Implementation

We have implemented the Nettle language as an domain-specific embedded language (DSEL) hosted in Haskell. The embedding confers significant benefits on Nettle, the most important of these being safe, comprehensive and uniform abstraction mechanisms and powerful calculation abilities. The examples in Sections 3, 4, and 5 illustrate how we can take advantage of these to write policy in a high-level and safe way.

We also take advantage of Haskell's type system, in particular its support for phantom types, generalized algebraic data types, type classes, and existentially qualified types to ensure that Nettle programs satisfy a number of logical constraints. An example of this is the design of the *RoutePred a* datatype, which carries a phantom type indicating the type of routes being predicated over. This phantom type prevents users from combining predicates over incompatible route types. For instance, it would be incorrect to intersect a predicate over static routes with a predicate over BGP routes, and our combinators appropriately prevent this. As more protocols are added to Nettle, it will become increasingly important to ensure predicates are constructed in sensible ways.

The phantom type of *RoutePred a* is used again when constructing route redistribution policies. As explained above, route redistribution injects some subset of routes, denoted by a route predicate, from one protocol to another. A route redistribution policy is then essentially specified by naming the exporting and importing protocols as well as a predicate over routes of the exporting protocol.

A network configuration may include several redistribution policies and we would like to collect these into a single list. However, since each of these policies may have predicates of different types, we need to hide the types in a quantified type. On the other hand, the Nettle-to-XORP compiler will need to know the exporting and importing protocols in order to generate correct code. To accomplish this we need to package the route redistribution policy with values indicating the exporting and importing protocols. In doing this, we want to ensure that these values correspond with the predicates appropriately. We accomplish all this by creating a type, whose values represent protocols and whose type also carries the protocol information:

```
data ProtocolValue a where
  BGPProtocol    :: ProtocolValue BGPT
  StaticProtocol :: ProtocolValue StaticT
```

We then use this in our definition of route redistribution policies:

```
data RedistPolicy = forall a b
  RedistPolicy (RoutePred a) (ProtocolValue a) (ProtocolValue b)
```

With these types, we can hide the predicate type in the redistribution policy so that it can be collected in a list, while providing values which can be used by the compiler by pattern matching against the constructors of the *ProtocolValue a* type. The phantom types ensure that the compiler will never be given an incompatible protocol value and route predicate. For example, *RedistPolicy* (*taggedWith* (1000:::200)) *BGPProtocol StaticProtocol* type checks, whereas *RedistPolicy* (*taggedWith* (1000 ::: 200)) *StaticProtocol BGPProtocol* does not.

Furthermore, we take advantage of Haskell's type class mechanism to implement overloaded operators, thereby simplifying the syntax of Nettle. Two examples of this are the sequential composition operator $\triangleright$ and the Boolean operators $\vee$ and $\wedge$. Sequential composition is used to denote concatenation of regular expressions as well as function composition of route modifiers. In this case we have made the appropriate data types instances of the *Data.Monoid* type class and made $\triangleright$ synonymous with the *mappend* function of this type class. The Boolean operations are defined in the *Boolean* type class and both route predicates and route predicate-valued functions are made instances of this, where for the latter the operations are defined pointwise, as follows:

```
class Boolean b where
  (∨)   :: b → b → b
  (∧)   :: b → b → b
  all   :: b
  none :: b
instance Protocol p ⇒ Boolean (r → RoutePred p) where
  b1 ∨ b2 = λr → b1 r ∨ b2 r
  b1 ∧ b2 = λr → b1 r ∧ b2 r
  all       = const all
  none    = conset none
```

7 Related Work

In analyzing BGP policy, Caeser and Rexford [12] emphasize the need for languages which express BGP policies more directly and allow for the expression of common policy patterns and Ramachandran [13] and Griffin [9] make contributions in how such languages should be constructed. Several other efforts, including the path vector algebra approach of Sobrinho [14] and Griffin [15] and the "Declarative Networking" approach of Loo et al [16] are promising approaches to this problem. These approaches differ from Nettle's approach in that they give languages for expressing both a protocol and a policy, whereas Nettle focuses solely on the kinds of policies supported by the current BGP protocol. In that regard, Nettle is much more closely related to configuration and policy languages such as XORP and RPSL, and in the following sections we describe the relationship of Nettle to those languages.

7.1 XORP

XORP routers are configured using a XORP configuration file. Unlike Nettle, which describes an entire network in a single program (though not necessarily a single file), XORP requires one file per router, and these files contain both router details such as details about network interfaces, and BGP routing policy. Policy is divided into "import" and "export" policy, which are similar to Nettle's usage and advertising policy. These import and export policies in turn consist of a sequence of guarded actions, similar to Nettle.

Xorp provides some limited facilities for naming and reusing elements. For example, it provides a construct for creating and naming sets of network prefixes that can be referred to in policies and it provides the ability to name a conjunction of basic predicates (XORP calls these "policy subroutines") for reuse. In contrast, Nettle provides much more extensive ability to name arbitrary policy components, as the above tutorial has demonstrated. The following Nettle example illustrates naming route modifiers, a simple form of naming which is currently not possible in XORP:

$$m1 = tag\ (5000 ::: 130)$$
$$m2 = prepend\ 65000$$
$$m3 = unTag\ (5000 ::: 90)$$
$$f = m1 \ \triangleright\ m2$$
$$g = m1 \ \triangleright\ m3$$

Even more importantly, XORP does not provide any functional abstractions. This prevents users from expressing their policy patterns in a reusable way and prevents users from designing their own higher-level policy languages or policy calculation methods.

7.2 RPSL

Routing Policy Specification Language (RPSL) is a stand-alone BGP policy language which, like Nettle, takes a network-wide view of BGP configuration. Like XORP it provides several constructs for naming policy elements, such as sets of routes and filters. Like XORP it does not provide comprehensive naming or functional abstraction.

Like Nettle, RPSL is vendor neutral and can be used to generate router configuration files. The *RtConfig* tool[17] generates a router configuration file from RPSL policy and script file containing router-specific information. Essentially, the RtConfig script file is a vendor-specific router configuration file with special commands inserted which instruct RtConfig to insert information from the RPSL policy. While this does manage to separate router specific details from BGP policy, it has the disadvantage that router-specific details are external to the language and there are therefore no mechanisms for reusing parts of these router-specific configurations or abstracting the patterns which generate them. In contrast, Nettle separates policy from router details, yet embeds both within a single language, Haskell, which gives users a uniform tool for manipulating both policy and router details.

8 Limitations and Future Work

Nettle currently only supports BGP and static routing. In order to be a practical alternative to vendor-specific router configuration scripts, Nettle will need to add support for the most common IGPs, such as RIP and OSPF. Nettle is also missing some BGP features commonly supported on all major platforms, including setting and testing the MED attribute and supporting route aggregation.

More significantly, while Nettle provides greater expressiveness than other routing configuration languages, it is still fairly low-level and does not fully address the issue of configuration correctness. For example, in the extended example of Section 4, the intended network behavior and the assumptions made of the network are expressed informally. The BGP configurations which achieve these objectives do not directly express the intentions, and the correctness of the configurations can only be judged with respect to those intentions. Furthermore, the assumptions about the network are essential in reasoning about the correctness of the configurations. We conclude that further work is needed in order to address the issue of policy correctness.

Acknowledgements. Thanks to Vijay Ramachandran, whose work on network routing policy inspired this effort. Thanks also to the anonymous reviewers who supplied helpful feedback on an earlier draft of this paper. This research was supported in part by NSF grant CCF-0728443 and DARPA grant STTR ST061-002 (a subcontract from Galois, Inc.).

References

1. Rekhter, Y., Li, T., Hares, S.: A Border Gateway Protocol 4. Internet Engineering Task Force (2006)
2. Alaettinoglu, C., Villamizar, C., Gerich, E., Kessens, D., Meyer, D., Bates, T., Karrenberg, D., Terpstra, M.: Routing Policy Specification Language (RPSL). Internet Engineering Task Force (June 1999)
3. Peyton Jones, S., et al.: The Haskell 98 language and libraries: The revised report. Journal of Functional Programming 13(1), 0–255 (2003)
4. XORP, Inc.: Extensible Open Routing Platform, XORP User Manual, Version 1.6 (January 2009)
5. Varadhan, K., Govindan, R., Estrin, D.: Persistent route oscillations in interdomain routing. Computer Networks 32(1), 1–16 (2000)
6. Mahajan, R., Wetherall, D., Anderson, T.: Understanding BGP misconfiguration. SIGCOMM Comput. Commun. Rev. 32(4), 3–16 (2002)
7. Hudak, P.: Building domain specific embedded languages. ACM Computing Surveys 28A (December 1996) (electronic)
8. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of Fifth International Conference on Software Reuse, pp. 134–142. IEEE Computer Society, Los Alamitos (1998)
9. Griffin, T.G., Jaggard, A.D., Ramachandran, V.: Design principles of policy languages for path vector protocols. In: SIGCOMM 2003: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 61–72. ACM, New York (2003)

10. Zhang, R., Bartell, M.: BGP Design and Implementation. Cisco Press (2003)
11. Gao, L., Rexford, J.: Stable internet routing without global coordination. SIGMETRICS Perform. Eval. Rev. 28(1), 307–317 (2000)
12. Caesar, M., Rexford, J.: BGP routing policies in isp networks. IEEE Network 19(6), 5–11 (2005)
13. Ramachandran, V.: Foundations of Inter-Domain Routing. PhD thesis, Yale University (May 2005)
14. Sobrinho, J.L.: Network routing with path vector protocols: theory and applications. In: SIGCOMM 2003: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 49–60. ACM, New York (2003)
15. Griffin, T.G., Sobrinho, J.L.: Metarouting. In: SIGCOMM 2005: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 1–12. ACM, New York (2005)
16. Thau, B., Joseph, L., Hellerstein, M., Stoica, I., Ramakrishnan, R.: Declarative routing: Extensible routing with declarative queries. In: Proceedings of ACM SIGCOMM 2005 (2005)
17. Meyers, D., Schmitz, J., Orange, C., Prior, M., Alaettinoglu, C.: Using RPSL in Practice. Internet Engineering Task Force (August 1999)