Specification of policy languages for network routing protocols in the Bellman-Ford family

Philip J. Taylor

University of Cambridge
Computer Laboratory
King's College

September 2011

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Specification of policy languages for network routing protocols in the Bellman-Ford family

Philip J. Taylor

Summary

Routing in large networks relies on protocols that find paths satisfying complex policy constraints. Protocols such as BGP are based on variants of the Bellman-Ford shortest paths algorithm but have extended far beyond this simple theoretical foundation, leading to bugs and stability problems as they have grown over time.

Algebraic routing provides a new theoretical framework for policy languages that can model much of the behaviour of these protocols, while also allowing proofs of correctness and stability. Metarouting aims to build practical routing protocols based on the theory of algebraic routing. Policy languages are specified in a carefully-designed metalanguage, where their algebraic correctness properties can be computed based on the constructive definition of the metalanguage, and they can also be compiled into efficient executable code. The challenge is to provide a useful amount of expressivity while still supporting correctness proofs.

In this dissertation we build and explore some core components of the metarouting implementation. First we give a detailed definition of the metalanguages used by the compiler, improving on previous publications. Next we implement a generalised version of the Quagga routing software that can be linked with these compiled policy languages to produce a complete and usable protocol implementation. This provides both a powerful model for understanding routing, and an implementation that can be used for rapid experimentation with new policy languages.

There is still a large gap between the complex realities of network routing and the model of algebraic routing. To reduce this gap and increase the applicability of metarouting, we develop extensions to model and to implement support for policy that is applied separately on export and import interfaces, and for policy expressed as route maps that can perform arbitrary computations without compromising the safety guarantees provided by the algebraic routing theory.

Acknowledgments

Foremost thanks go to my supervisor, Tim Griffin. Additional thanks go to the meta-routing group for valuable discussions and feedback, in particular John Billings, Alex Gurney, M. Abdul Alim and Vilius Naudžiūnas, and also to Jeroen De Ridder for feedback on this document, and to the Engineering and Physical Sciences Research Council (EPSRC) for support (Grant EP/F002718/1).

Contents

Chapter 1

Introduction

1.1 Internet routing

Internet routing aims to solve the problem of efficiently transmitting data packets from any device connected to the network anywhere in the world to any other device. The current solution has evolved and grown over the past two decades – today it consists of about 40,000 individually-operated networks linked together in a complex mesh, with over a third of a million distinct routes to over two billion endpoints, and still growing with no sign of slowing down [Hus11]. It has to respond continually to changes in the network as components are added or removed or fail unexpectedly, as well as to cope with the commercial demands of a growing multi-billion-dollar industry [Del11] where competitors are required to work together to maintain connectivity throughout a single global network.

Much of the software and network protocols that are still in use today originated in the late 1980s and early 1990s. The Border Gateway Protocol (BGP) – the protocol that joins every individual network (known as an *autonomous system* or AS) into the global Internet – published its first version in 1989 [LR89] and its last major version, BGPv4, in 1994 [RL94], and has been incrementally extended many times since then.

Although this system has scaled remarkably well to modern demands, it has several fundamental problems. For a start, it is reasonable to expect that if a network implements a routing protocol and does not violate that protocol's checkable requirements, and the network remains stable for a sufficient period of time, then the protocol will converge to a state that provides good routes through the network. However, it has been shown that BGP does not have this property: it can suffer from persistent oscillations [VGE00, GW02], or can get stuck in undesirable states [GH05]. Many of these bugs were not anticipated in the original design of BGP and were discovered only after debugging problematic behaviour in large deployed networks – in one reported case an oscillation lasted for five days and made up 95% of the ISP's BGP traffic [WJA04] –

by which time it was too late to fix the protocol design without breaking compatibility. The research into these problems has suggested various network configuration guidelines to avoid encountering them, but this is at best an unsatisfactory solution relying on constant vigilance by network operators and protocol designers; though there has been some work on static analysis tools to detect a subset of problems in BGP [FB05], determining global convergence properties of a network configuration is in general an NP-hard problem [GW99]. The set of stability problems itself is not stable – the typical process of designing and extending protocols still consists of writing RFCs that describe behaviour in a router at the mechanistic level of bytes and of algorithm implementations, providing little protection against the introduction of further stability bugs and making the behaviour very hard to analyse. Bornhauser et al. [BMH11] explore the causes of routing anomalies in a component of BGP named iBGP, particularly in widely-used extensions beyond the original specification, and note that "drafts for new iBGP extensions come along with similar conceptual defects": there is not yet a sufficient understanding in the field to stop the continued introduction of new problems.

Furthermore, today's implementation of Internet routing provides very limited flexibility. Routers typically implement a small range of routing protocols: BGP as the *external gateway protocol* (EGP) that connects autonomous systems together; and RIP [Mal98], OSPF [Moy98], IS-IS [iso02] or EIGRP [AGLAB94] as choices for the *internal gateway protocol* (IGP) that is used for routing within a single AS. IGPs are typically approximations of *shortest-path routing*, automatically finding paths through the network that optimise latency or resource usage, whereas EGPs must provide *policy-based routing* where network administrators can restrict routing decisions based on commercial relationships or other external requirements. As individual networks have grown in complexity, often by absorbing other networks and often spanning the globe, some ASes have outgrown the limited selection of IGPs and been forced to press the more powerful BGP into service as a policy-rich IGP [WMS05, Ch. 3], a use for which it was not designed and is not ideally suited.

We believe it is valuable to take a step back from the current solution to Internet routing and examine its foundations from a fresh perspective. *Metarouting* [GS05] is an approach that splits the concept and implementation of a routing protocol into two distinct components: a *routing language* that is used to encode policy (sometimes referred to as a *policy language*), and a *routing algorithm* that computes paths based on policies from any routing language. In this model, routing languages are implemented using the theory of *algebraic routing* [Sob02, Sob03] to address the problem of protocol correctness: despite a protocol typically relying on complex distributed asynchronous computations, the problem of proving that it will converge to a correct solution is reduced to the (often much simpler) problem of proving the routing language exhibits certain algebraic properties, given a prior understanding of the routing algorithm that

is used. Metarouting builds on this theory by providing a language to constructively define complex algebras from simple well-understood components, such that the algebraic properties necessary for correctness can be automatically inferred. Finally, metarouting provides a practical implementation of this language for defining algebraic routing languages – the *metalanguage* – allowing users to compile concise language specifications into running routing-protocol code. This implementation begins to address the problem of limited flexibility in routers, by greatly simplifying the process of creating protocols with distinctive new forms of policy, allowing rapid experimentation while the algebraic underpinnings guarantee correct protocol operation; a protocol constructed in this way will have the most serious convergence problems identified at design time. Even when protocols do not use the metarouting implementation in practice, an understanding of the metarouting system still provides insight into the important tradeoffs and decisions in any new routing protocol design.

In this dissertation we develop the metarouting approach to better match the functionality and policy-richness provided by current routing protocols, focusing specifically on policy-based *vector routing protocols* that are based on the Bellman-Ford algorithm, such as BGP, RIP and EIGRP. There are many gaps between the foundational algebraic routing theory and the practical needs of Internet routing. We aim to bridge some of the most vital of those gaps. Firstly, this bridging demonstrates the applicability of the theoretical approach to a wider range of routing problems: it allows algebraic techniques to be used for analysing more complex protocol behaviours, and allows the theoretical correctness guarantees to be more directly applicable to implementations. Secondly, it improves the viability of the metarouting implementation, letting its routing language specification and compilation process be used for developing increasingly complex routing protocols.

1.2 Metarouting system overview

Metarouting consists of a large number of interacting components. Some have been outlined or detailed in earlier published work, though not all, and most have evolved since earlier publications. Chapters 2 and 3 aim to give a new, consistent detailing of the parts that are relevant to this dissertation. This section gives an overview of all the components and their published work, starting with the core and gradually adding the additional layers that make up the complete picture of metarouting, and discusses the motivation for including each component into the system.

As the starting point for this system, algebraic routing as developed by Sobrinho [Sob02, Sob03, Sob05] models a routing protocol as a *routing algebra* to express all the aspects of policy, combined with a generalised *routing algorithm* that performs the necessary computation steps, to produce a routing protocol whose convergence behaviour can

be guaranteed by the algebra's properties. In particular, if the algebra is *increasing* (or "monotonic" in Sobrinho's terminology) then the algorithm will always reach a stable solution in any network configuration. Importantly, this is an if-and-only-if relationship: if the algebra is *not* increasing, then there is some network configuration in which the algorithm will fail to converge. This provides clear and strongly-justified constraints on policy. The use of algebras as the formalism for policy makes available a wide range of existing mathematical literature for understanding its properties and behaviour, including work on generalised pathfinding with algebras known as semirings [Car79, GM84, GM08].

There are some routing algebras that could guarantee convergence in a subset of network topologies and configurations, for example when networks are connected in a strict hierarchy and not an arbitrary mesh. Internet routing is often viewed as a hierarchy – Gao and Rexford [GR01] show that the hierarchy produced by common commercial relationships can provide stability in the Internet; Xia and Gao [XG04] infer these relationships based on observed Internet routing data; the proposed HLP protocol [SCE+05] uses an algorithm optimised for this hierarchy. However, there is nothing enforcing this and the number of exceptions grows as the Internet becomes more complex; it has been observed that the Internet has "migrated from a fairly tree-like connectivity graph to a meshier style" [DD10] over time. This lack of a strict hierarchy, together with the NP-hardness of detecting convergence problems in arbitrary networks [GW99], motivates our desire to provide correctness as a fundamental part of the routing protocol that is not dependent on having networks follow an unenforceable set of rules. We therefore do not include network topologies in our design of the metarouting system, focusing solely on topology-independent properties.

On the implementation side, protocols today are typically monolithic implementations of the standard RFCs for BGP and RIP and so on, written in C or C++. These include open source software such as Quagga [qua] and XORP [HKG+05], and commercial devices from companies such as Cisco [cis] and Juniper [jun]. Mapping these to any theoretical model is a difficult task of reverse-engineering, and requires extracting the semantics from prose and pseudo-code in the RFCs or (where they extend or diverge from the RFCs) from the implementations themselves. Earlier work introduced formalisms such as the Simple Path Vector Protocol [GSW99, GSW02] as greatly simplified approximations of BGP's operation, and extracted correctness properties such as the customer/provider/peer guidelines [GR01] based on analysing the typical usage of BGP policy.

Figure 1.1 illustrates the relationships between components in this model; we will extend this diagram to show how new components interact and form the complete system. On the right, the algebraic routing theory lets us understand protocols as a combination of algebra and algorithm. On the left, the implementation is a single component that conflates these tasks.

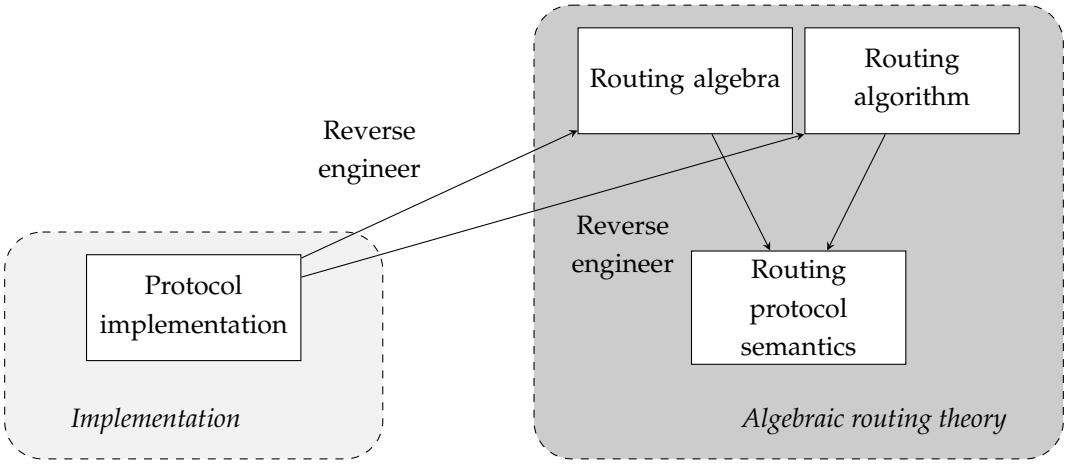


Figure 1.1: Overview diagram of the initial approach to algebraic routing.

The reverse engineering provides an important starting point for understanding the protocols, but is laborious and far from robust given the complexity of the analysis and the infeasibility of verifying the implementation against the theory.

We want to reduce the gap between implementations and the algebraic theory to make them easier to relate. The first incremental change we make to this model is to split the protocol implementation into two clearly separate components, one for the routing language and one for the algorithm, which are linked to produce the complete protocol, as in Figure 1.2.

In the current metarouting implementation the routing algorithm implementations are derived from the open source Quagga routing suite, which provides standard protocols such as BGP, RIP and OSPF. We have removed the default hard-coded policy behaviour, allowing arbitrary routing language implementations to be plugged in, giving a set of generalised algorithm implementations that we call *gQuagga*. In Chapter 4 we discuss the details of this generalisation process. The new algorithms can be considered instances of generalised distributed Bellman-Ford and Dijkstra algorithms, though in practice they retain a strong flavour of the protocol from which they were derived, and so we call the individual generalised algorithms *gBGP*, *gRIP*, *gOSPF* and so on. Other algorithm implementations – for example based on XORP, or offline network simulators – could also be used, though for simplicity we will only consider algorithms with the same API as gQuagga.

Routing language implementations in this case are executable C or C++ code libraries that can plug into the gQuagga algorithms. In principle these could be arbitrary hand-written code – the only requirement is that they implement the language–algorithm API detailed in Chapter 4. We use C++ simply because all practical routing protocol implementations are written in C or C++ and this minimises the friction of combining the algorithm and language code. (If we were to use routing algorithms written in

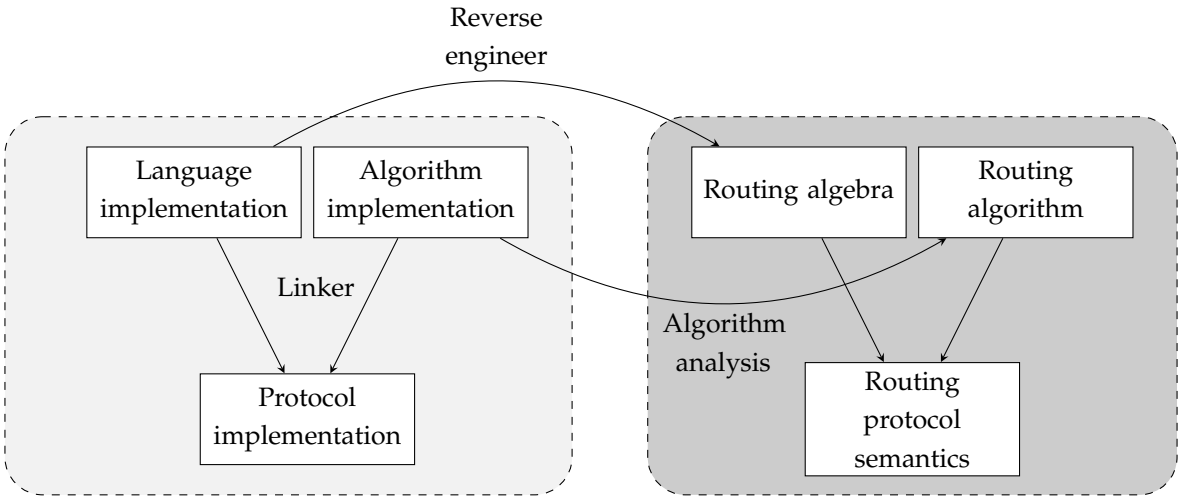


Figure 1.2: Overview diagram with algorithm/language split in implementations.

a very different programming language, the routing languages should likely use that same programming language, but there would typically be little fundamental difference from our current choice of C++.)

This algorithm/language split in the implementation makes it slightly easier to map onto the theoretical model. The routing language code can be examined to determine what routing algebra it implements (if any), without worrying about the details of how it is mixed in with the algorithm code. Similarly, we can analyse gBGP and gRIP to determine how closely they implement an idealised distributed Bellman-Ford algorithm, without depending on the details of the protocol's policy design.

However, this change still does not provide a straightforward way to relate the routing language to a routing algebra that can be checked for correctness. It also does not simplify the task of implementing a routing language – there is likely to be a large amount of boilerplate code that the algorithm requires (e.g. most of the language's data types need code to send and receive over the network, read input from configuration commands, output to debug logs, handle explicit memory management, etc.), greatly increasing the cost and the likelihood of bugs when experimenting with new policy languages.

This separation of language/algorithm implementation is therefore just a stepping stone towards the next stage. To address the difficulties we raise routing languages to a higher level of abstraction than C++, by developing a domain-specific language that we call the *executable routing language*[1] (ERL). Figure 1.3 shows how this fits into the system.

This new language has an explicit algebraic semantics: any ERL specification can be

[1] Since this is a language for describing an executable form of routing languages, we should probably call it the executable routing *meta*language, but the ERL name has stuck.

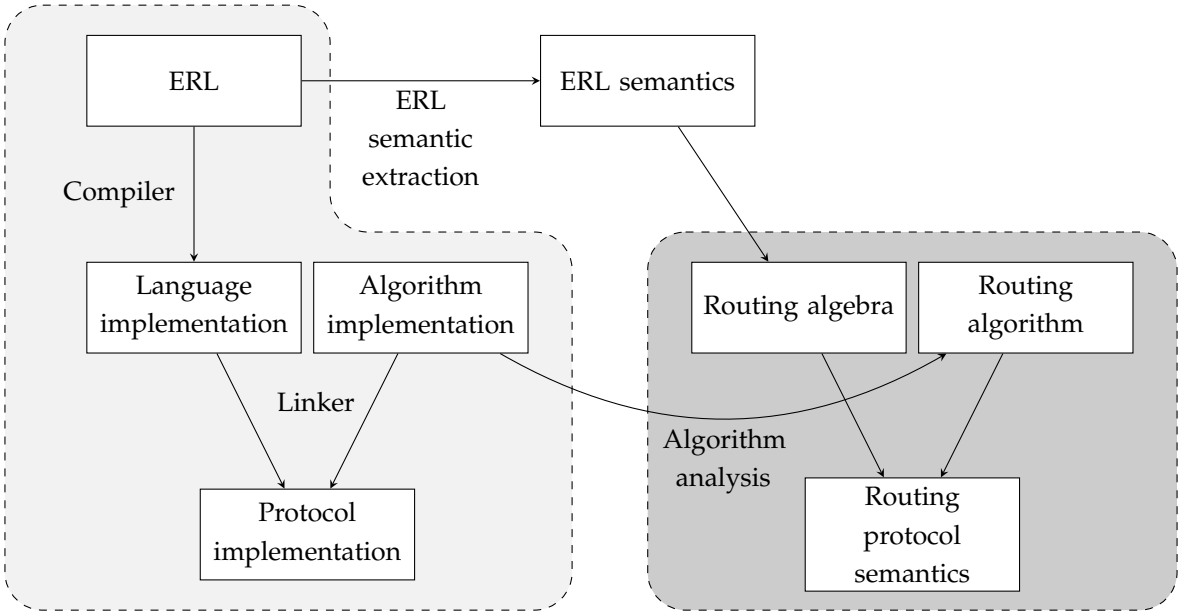


Figure 1.3: Overview diagram with routing languages implemented in ERL.

mechanically mapped onto a routing algebra, eliminating the need to reverse engineer the behaviour from C++ code. It also eliminates boilerplate code: ERL is a declarative language from which an ERL-to-C++ compiler can generate all the code required by the gQuagga API, greatly reducing the effort involved in developing a new routing language. The language also increases implementation quality compared to manual coding in C/C++ by eliminating whole classes of potential bugs, for example by automatically handling memory allocation and by reusing a library of robust functions for parsing complex data types from network packets.

ERL has much less expressivity than C++. It is also unable to express many classes of routing algebras that could be defined mathematically. However, it has been designed to be capable of expressing a wide range of the routing algebras that are interesting for network routing; it turns out that a small collection of primitives (sets, minimal sets, lists, simple (duplicate-free) lists, bounded integers, lexicographic products, etc.) are sufficient for most of the functionality of current routing protocols and for many novel ones. Billings [Bil09] defines an earlier version of the ERL language (there named "IRL") and explains the process of compiling to efficient C++ code. We reformulate ERL in Chapter 3 but the principle of the language and its compilation process remain the same.

Although the semantics of ERL let us derive the routing algebra corresponding to any expressible routing language, they do nothing to help us determine its algebraic properties. These are the properties that are crucial for determining whether the completed protocol will converge to a correct solution, and given an arbitrary algebra it is often not easy – and sometimes impossible – to either prove or disprove that it has a partic-

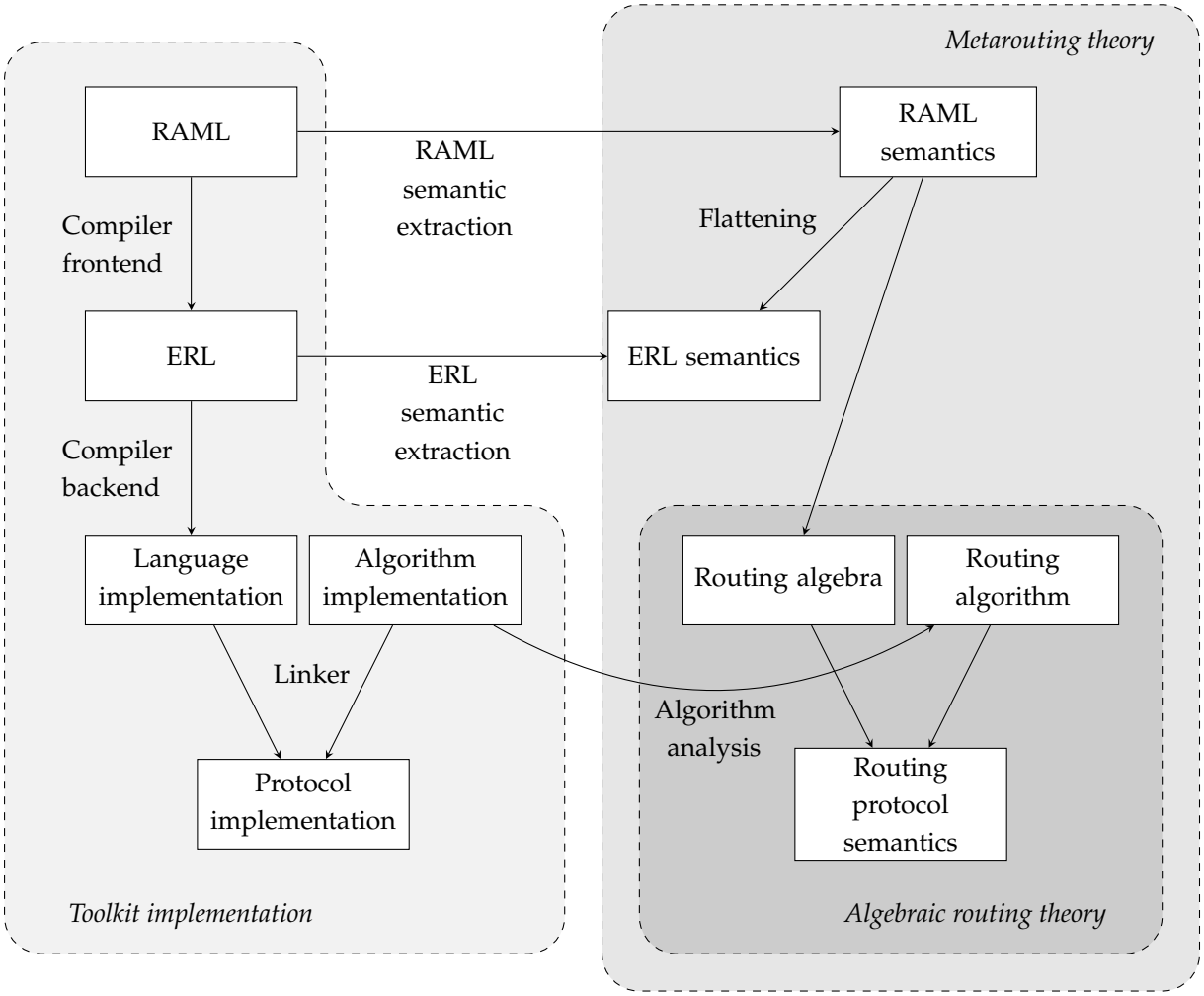


Figure 1.4: Overview diagram with routing languages implemented in RAML.

ular property. To solve this we add another level of abstraction, in the form of a new language built on top of ERL.

In this completed model of the metarouting system, shown in Figure 1.4, routing languages are specified in the *Routing Algebra MetaLanguage* (RAML). Although ERL can be translated into a routing algebra, this new language is much more tightly linked to the algebras: the semantics of a routing language specification written in RAML is the algebra plus a complete set of algebraic properties. RAML is carefully designed so that these properties can be derived from a language specification by if-and-only-if rules, guaranteeing that it will either confirm a language is safe to use, or else will be able to say exactly why a language is unsafe.

ERL is now demoted to being an *intermediate language* emitted by a RAML-to-ERL compiler. It is still a valuable abstraction, simplifying the definition and implementation of RAML: ERL is (by design) a good match for the executable semantics of RAML, and sufficiently more expressive that we can often extend the RAML language with-

out having to make any changes to ERL or to the ERL-to-C++ compiler. This allows us to understand and extend RAML in isolation from the messy details of executable C++ code. We make heavy use of this extensibility in Chapter 5.

Given a RAML specification, we can extract its RAML semantics – a mathematical definition of a routing algebra, plus a set of algebraic properties that it satisfies – based on the definition of semantics in Chapter 3. Alternatively we can compile the RAML to ERL, then extract its ERL semantics directly – an expression defining a routing algebra in a lower-level form, but with no algebraic properties. If we 'flatten' the RAML semantics by dropping its properties, we end up with an equivalent algebra to the ERL semantics (though perhaps expressed in a slightly different form); this requires that our definitions of RAML semantics, ERL semantics, and RAML-to-ERL compilation are consistent with each other.

The design of RAML is challenging due to the desire to provide if-and-only-if rules for the algebraic properties of any RAML expression, while also maximising its expressivity. The early design of metarouting by Griffin and Sobrinho [GS05] included a small set of basic routing algebras, a lexicographic product constructor, and a few more specialised constructors such as the scoped product (to model the inter-AS and intra-AS distinction of BGP routing). Later work by Gurney and Griffin [GG07] has shown that scoped products can be implemented using lexicographic products and a few other primitive constructors, including full property inference. This process of replacing specialised operations with more basic generic operations has allowed RAML to become more powerful and expressive while also simplifying its definition by minimising special cases.

The design of RAML is ongoing work, so in this dissertation we will focus on a subset instead of attempting a comprehensive definition. Billings [Bil09] defines a version of the RAML syntax; as with ERL we will reformulate this definition in Chapter 3 to better fit our presentation of the system's architecture, but we do not consider the implementation of the RAML compiler or the process of computing its property inference rules. Current work by Naudžiūnas [NG11, Nau11] implements a form of RAML in the Coq theorem prover [BC04]. Coq can automatically verify the correctness of the inference rules (avoiding the risk of human error when proving rules then transcribing them into the code), and an implementation of the RAML compiler can be automatically extracted as self-contained OCaml code. The RAML compiler does not use the theorem prover itself: it is not dynamically proving theorems about a particular routing algebra specification, it is merely applying the general rules which were verified in advance by the theorem prover.

All of these components fit together to complete our picture of the metarouting system. From an implementation perspective, the result is a tool which allows a user to write a high-level declarative specification of a routing language in RAML and then either

be told why it is unsafe to use in practice, or get a complete usable routing protocol that can be deployed and run on any Quagga-compatible router. From a research perspective, the result is a well-founded theoretical basis for analysing and discussing the behaviour of routing protocols, and an exploration of the tradeoffs between safety and policy expressivity.

1.3 Chapter outline

Chapter 2 summarises the relevant background information, including defining the concepts of algebraic routing that we build on.

Chapter 3 introduces a new, consistent and detailed definition of the syntax and semantics of ERL and RAML.

Chapter 4 develops the gRIP and gBGP algorithm implementations, to demonstrate that we really can separate out a lot of the implementation details from the routing language, and that the language/algorithm concept is a useful abstraction allowing us to use and analyse each part independently. It also considers what parts of existing protocols are 'language' and what parts are 'algorithm'; in some cases there are several different ways to perform the split.

Chapter 4 also describes the details of binding a language implementation (the output of the ERL-to-C++ compiler) to the algorithm implementation. The binding is briefly described by Billings [Bil09] from the routing language perspective; here we discuss it from the routing algorithm perspective, which introduces an additional set of concerns, to give a more complete picture. (The design and implementation of this interface was performed by the author in conjunction with John Billings.) The binding needs to provide the bridge between the provably-correct theoretical model of language, and the low-level messy world of algorithm code, in a way that is both theoretically sound (by avoiding introducing a lot of non-analysable complexity that may break the applicability of correctness properties to the implementation) and practically sound (efficient and compatible with current algorithm implementations).

Chapter 5 looks at the nature of router configuration in vector protocols, where the specification and computation of policy is split across multiple routers. Because of the goal to keep the algorithm binding simple (so it doesn't introduce correctness bugs), we put the complexity in the language side where we can analyse it theoretically. We add simple extensions to the binding and algorithm implementations to handle the new form of language. This does not require any modifications to the algebraic routing theory: the new language model can be mapped onto the original model, allowing all the existing theoretical results to be reused. Given our configuration model, some new protocol features are relatively straightforward to implement: we add a form of

outbound route filtering that is more general than the current ORF work on BGP, to demonstrate the utility of the model.

Chapter 6 extends the system to support *route maps*, as an expressive form of policy that is widely supported by standard routing protocol implementations today. This changes the structure of computations performed by the generic routing algorithms, so we explore the consequences this has on algebraic properties.

In Chapter 7 we demonstrate how these extensions to the metarouting model work in practice by running an example through the whole system.

1.4 Contributions

The main contribution of this dissertation is to demonstrate that the concept of metarouting can be applied not just to an abstract shortest paths algorithm but to real vector routing protocols, incorporating much of the complexity that is provided by current implementations while building on a strong theoretical framework that provides flexibility and safety guarantees that are missing from network routing. In particular, this consists of:

- An analysis of several vector protocols (primarily RIP, BGP and EIGRP) based on reverse engineering the protocol specifications and implementations, to provide an understanding of their features and their behaviour from the perspective of algebraic routing. This highlights a number of areas where previous work on metarouting has been insufficient for modelling vector protocols.
- An implementation of generalised vector routing algorithms, based on Quagga's RIP and BGP protocols, that can execute a compiled routing language to set up the routing tables of a network.
- An extension of the metarouting theory and implementation to support the specification of policy languages that have separate configuration data and perform separate computations at the export and import interfaces of routers when running a Bellman-Ford–style algorithm.
- An analysis of several current protocol implementations to determine the common functionality and full extent of the varied features that are referred to as route maps, so that we can understand the scope of the term, and to identify their different approaches to designing route map languages.
- An extension of the metarouting theory to support functionality similar to route maps as a more expressive method for configuring policy on a router.

Chapter 2

Background

This chapter describes the technologies, concepts and terminology that the dissertation builds on.

2.1 Shortest paths problem

The fundamental mathematical model we use for network routing is the shortest paths problem. In particular we are interested in the single-source shortest paths problem over directed graphs with no negative edge weights [CLR92]. Figure 2.1 illustrates such a graph. In general a graph $G = (V, E)$ consists of a set of vertices V, and a set of edges $E \subseteq V \times V$, along with an edge weight function $w : E \to \mathbb{N}$. (We will sometimes use the term *node* for vertices and *arc* for edges.) An edge (v, v) from a vertex to itself is not permitted.

A path $p = [v_0, v_1, \ldots, v_k]$ (where each adjacent pair of vertices in the path is in E) has path weight $w(p) = w(v_0, v_1) + w(v_1, v_2) + \ldots + w(v_{k-1}, v_k)$. A zero-length path $p = [v_0]$ has weight 0.

The shortest path weight $\delta(u, v)$ from vertex u to v is $\min(\{w(p) \mid p \in \mathcal{P}(u, v)\})$ where $\mathcal{P}(u, v)$ is the set of all paths $p = [u, \ldots, v]$ in the graph, or ∞ if there are no such paths. A particular path $p \in \mathcal{P}(u, v)$ is a (not necessarily unique) shortest path if $w(p) = \delta(u, v)$.

We can represent a graph as an adjacency matrix $\mathbf{A}$, where $\mathbf{A}(u, v) = w(u, v)$ if $(u, v) \in$

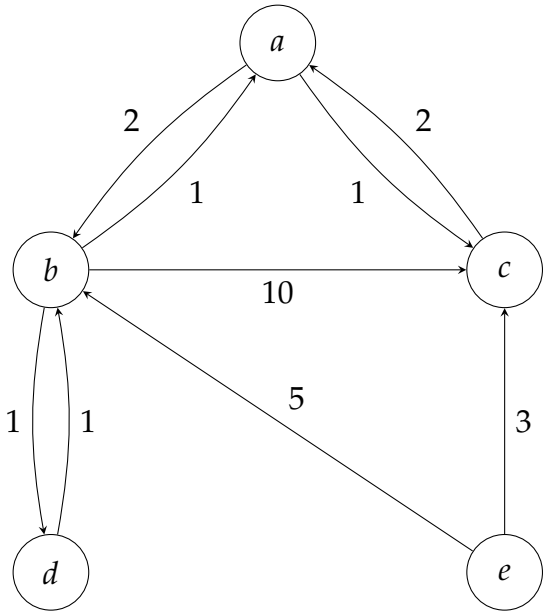


Figure 2.1: A directed graph with integer edge weights.

E, and ∞ otherwise. Figure 2.1 has the adjacency matrix

$$\mathbf{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} \infty & 2 & 1 & \infty & \infty \\ 1 & \infty & 10 & 1 & \infty \\ 2 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty \\ \infty & 5 & 3 & \infty & \infty \end{bmatrix} \end{array}.$$

The solution to the single-source shortest paths problem with source u is a vector $\mathbf{D}$ where $\mathbf{D}(v) = \delta(u, v)$. There are a number of well-known algorithms that can be used to find solutions, such as Dijkstra's algorithm [Dij59] (the basis of link state routing, used by the OSPF and IS-IS routing protocols) and the Bellman-Ford algorithm (the basis of vector routing, used by the RIP and BGP protocols). This dissertation focuses exclusively on variants of the Bellman-Ford algorithm.

2.1.1 Bellman-Ford

The algorithm known as Bellman-Ford was originally developed by Bellman [Bel58] and by Ford and Fulkerson [FF62]. It is typically described in pseudocode, as in Figure 2.2 (based on the version from Cormen, Leiserson and Rivest [CLR92]).

We can also express the algorithm as an iteration over an adjacency matrix $\mathbf{A}$ and a vector $\mathbf{X}$, which provides a different way to view its behaviour. For our example graph

```
1  for each vertex v ∈ V[G]
2        do x[v] ← ∞
3  x[d] ← 0
4  for i ← 1 to |V[G]| − 1
5        do for each edge (u, v) ∈ E[G]
6              do if x[v] > x[u] + w(u, v)
7                    then x[v] ← x[u] + w(u, v)
```

Figure 2.2: Bellman-Ford algorithm pseudocode. Input is graph G and source vertex d; output is vector x of shortest path weights to every vertex.

with source vertex d, we start with

$$\mathbf{X}^{(0)} = \mathbf{I} = \begin{matrix} \\ d \end{matrix} \begin{matrix} \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{bmatrix} \infty & \infty & \infty & 0 & \infty \end{bmatrix} \end{matrix}$$

representing the best known distance from d to each other vertex as of step 0. We then perform the iteration

$$\mathbf{X}^{(k+1)} = (\mathbf{X}^{(k)} \cdot \mathbf{A}) \ \min \ \mathbf{I}$$

where the operator "$\cdot$" is based on the standard rules of matrix multiplication but with $\times$ replaced by $+$ and $+$ replaced by min:

$$(\mathbf{X} \cdot \mathbf{A})(i, j) = \min_{q \in V} (\mathbf{X}(i, q) + \mathbf{A}(q, j)).$$

(This is a variation of the matrix multiplication solution to the general all-pairs shortest paths problem [CLR92], restricted to a single row.) Depending on the order used by the iteration over edges in the Bellman-Ford algorithm, it will be performing an equivalent computation to this matrix method in each step. The BF algorithm may use a different order over edges, in which case it may produce different intermediate results (as updates to $x[v]$ may be read as $x[u]$ by a later step in the same iteration of the outer loop), but a property of the algorithm is that the final output will be the same regardless of order and therefore the same as the matrix method.

Applying this to our example gives

$$\mathbf{X}^{(1)} = \begin{matrix} \\ d \end{matrix} \begin{matrix} \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{bmatrix} \infty & 1 & \infty & 0 & \infty \end{bmatrix} \end{matrix}$$

$$\mathbf{X}^{(2)} = \begin{matrix} \\ d \end{matrix} \begin{matrix} \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{bmatrix} 2 & 1 & 11 & 0 & \infty \end{bmatrix} \end{matrix}$$

$$\mathbf{X}^{(3)} = \begin{matrix} \\ d \end{matrix} \begin{matrix} \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{bmatrix} 2 & 1 & 3 & 0 & \infty \end{bmatrix} \end{matrix}$$

$$\mathbf{X}^{(4)} = \begin{matrix} \\ d \end{matrix} \begin{matrix} \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{bmatrix} 2 & 1 & 3 & 0 & \infty \end{bmatrix} \end{matrix}$$

```
1  for each vertex v ∈ V[G]
2        do x[v] ← ∞
3  x[d] ← 0
4  for i ← 1 to |V[G]| − 1
5        do for each vertex v ∈ V[G]
6              do for each vertex u ∈ V[G] where (u, v) ∈ E[G]
7                    do if x[v] > x[u] + w(u, v)
8                          then x[v] ← x[u] + w(u, v)
```

Figure 2.3: Distributed Bellman-Ford algorithm pseudocode.

Given that no negative edge weights are allowed in our graphs, the Bellman-Ford algorithm will reach a stable solution after $|V| - 1$ iterations: $\mathbf{X}^{(|V|-1)} = \mathbf{X}^{(|V|-1)} \cdot \mathbf{A}$, and this will be equal to the shortest paths solution $\mathbf{D}$. In this example $|V| = 5$, so the solution is $\mathbf{X}^{(4)}$.

2.1.2 Distributed Bellman-Ford

In network routing, each vertex in the graph corresponds to a router and each edge corresponds to a communication link between routers.

If each router in a network of size N stores the complete adjacency matrix $\mathbf{A}$, the algorithm's per-router memory requirements are $O(N^2)$, leading to a scalability problem in large networks. Routing protocols based on Dijkstra's algorithm, such as OSPF, are typically restricted to small networks (with partitioning mechanisms to combine them into larger networks) due to this scalability issue. However, protocols based on Bellman-Ford can use a distributed variant of the algorithm in which the requirements on each individual router scale as $O(N)$, allowing larger networks to be handled efficiently, with the tradeoff of slowing the algorithm's execution by adding communication delays.

The distributed Bellman-Ford algorithm (DBF) [BG92, Sec. 5.2.4] is based on the observation that the original BF algorithm can be written as in Figure 2.3, and that the iteration on line 5 can be executed in parallel with every router processing the v corresponding to itself. In this case, router v will only need to know the element $x[v]$, plus $x[u]$ and $w(u, v)$ for each router u to which it has an edge. These edges correspond to network links, so v can easily determine these values by communicating directly with each u over the network. The amount of data stored and processed by v will scale with the number of routers to which it has an edge, so it is bounded by $O(N)$.

As a final modification, the loop on line 4 (which implies each router v will run lines 5–8 and then wait for every other router to finish before proceeding in lockstep) is re-

moved: every router repeats lines 5–8 forever and asynchronously. Although it is hard to tell how long the algorithm will now take to reach the solution, it will eventually converge to the same solution as the original Bellman-Ford algorithm. Bertsekas [BG92] gives a proof of convergence for the asynchronous DBF shortest-paths algorithm, while Sobrinho [Sob03] gives a proof for an asynchronous message-passing path vector protocol based on DBF that is generalised to certain routing algebras.

2.1.3 Counting to infinity

Although we stated that Bellman-Ford's iterations will reach a stable solution, this assumed we started with the given $\mathbf{X}^{(0)}$ containing infinities and 0. In a DBF implementation for use in network routing, we need to cope with dynamic changes to the network (vertexes and edges being added and removed). Restarting the algorithm from a clean slate after every minor change would cause significant performance problems in highly dynamic networks. Since most changes will be small and their effects will be localised, it is common to start a new iteration from the algorithm's state before the change: $\mathbf{X'}^{(0)} = \mathbf{X}^{(k)}$.

In our example, we will modify the adjacency matrix $\mathbf{A}$ by removing the edge from d to b,

$$\mathbf{A'} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} \infty & 2 & 1 & \infty & \infty \\ 1 & \infty & 10 & 1 & \infty \\ 2 & \infty & \infty & \infty & \infty \\ \infty & \underline{\infty} & \infty & \infty & \infty \\ \infty & 5 & 3 & \infty & \infty \end{bmatrix} \end{array},$$

and start the iteration with the state

$$\mathbf{X'}^{(0)} = \begin{array}{c} \\ d \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 2 & 1 & 3 & 0 & \infty \end{bmatrix} \end{array}.$$

This corresponds to d losing network connectivity after the routing algorithm had reached a stable state. Now we can run the matrix iteration again, giving the sequence shown in Figure 2.4.

In this case, the algorithm is *not* converging on a stable solution. This is the *counting to infinity* problem, caused by stale information circulating through the network. At each step, node a thinks the best route from d is through b while b thinks it is through a. Since d is disconnected and the true weights of shortest paths from it are ∞, this stale information is always preferred over the up-to-date true information. Different routing protocols take different approaches to resolving this problem, which we will discuss later. As long as this stale information is purged from the system eventually,

$$
\begin{aligned}
\mathbf{X'}^{(0)} &= d \begin{bmatrix} \overset{a}{2} & \overset{b}{1} & \overset{c}{3} & \overset{d}{0} & \overset{e}{\infty} \end{bmatrix} \\
\mathbf{X'}^{(1)} &= d \begin{bmatrix} \overset{a}{2} & \overset{b}{4} & \overset{c}{3} & \overset{d}{0} & \overset{e}{\infty} \end{bmatrix} \\
\mathbf{X'}^{(2)} &= d \begin{bmatrix} \overset{a}{5} & \overset{b}{4} & \overset{c}{3} & \overset{d}{0} & \overset{e}{\infty} \end{bmatrix} \\
\mathbf{X'}^{(3)} &= d \begin{bmatrix} \overset{a}{5} & \overset{b}{7} & \overset{c}{6} & \overset{d}{0} & \overset{e}{\infty} \end{bmatrix} \\
\mathbf{X'}^{(4)} &= d \begin{bmatrix} \overset{a}{8} & \overset{b}{7} & \overset{c}{6} & \overset{d}{0} & \overset{e}{\infty} \end{bmatrix} \\
\mathbf{X'}^{(5)} &= d \begin{bmatrix} \overset{a}{8} & \overset{b}{10} & \overset{c}{9} & \overset{d}{0} & \overset{e}{\infty} \end{bmatrix} \\
\mathbf{X'}^{(6)} &= d \begin{bmatrix} \overset{a}{11} & \overset{b}{10} & \overset{c}{9} & \overset{d}{0} & \overset{e}{\infty} \end{bmatrix}
\end{aligned}
$$

Figure 2.4: Demonstration of counting to infinity behaviour.

the BF algorithm is guaranteed to converge to the correct solution regardless of the initial state $\mathbf{X'}^{(0)}$.

2.2 Algebraic routing

In our description of the shortest paths problem, we represent path weights as values from the set $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ (where ∞ represents the absence of any path), and edge weights also as values from $\mathbb{N}^\infty$ (where ∞ represents the absence of an edge). A path weight is the sum of its edge weights, and the shortest path is the minimum of all path weights. The Bellman-Ford algorithm similarly uses the $+$ operator and either the min operator or $\leq$ order in defining its iteration to compute the shortest paths.

Algebraic routing is based on the shortest paths problem, but generalised to work over any algebra of a certain structure with certain algebraic properties. This section will introduce and define the algebraic concepts that are needed.

2.2.1 Types

The most fundamental component is *types*, which we consider to simply be sets of values. Common types are defined in Table 2.1.

Notation	Meaning
$\mathbf{1}$	Unit type, $\{1\}$
$\mathbb{Z}$	Integers
$\mathbb{N}$	Natural numbers (non-negative integers)
$\mathbb{N}^+$	Positive natural numbers
$\mathbb{N}^\infty$	Natural numbers plus infinity, $\mathbb{N} \cup \{\infty\}$
T^*	Finite lists, with elements of type T
T^*_{simp}	Simple lists, i.e. lists where no element occurs more than once
$S \uplus T$	Disjoint union, $\{\text{inl}(s) \mid s \in S\} \cup \{\text{inr}(t) \mid t \in T\}$
$\wp(T)$	All finite[1] subsets of T, i.e. sets whose elements are of type T
$\wp_\preceq(T)$	All finite minimal subsets of T, i.e. sets $S \in \wp(T)$ where $a, b \in S \Rightarrow a \not\preceq b$

Table 2.1: Definitions of common types.

2.2.2 Semigroups

Another important component of these algebras is the *semigroup*, written as

$$(S,\ \otimes)$$

and consisting of a *signature* type S and a binary operator $\otimes$ of type $S \times S \to S$ (that is, it takes two values of type S and returns one of type S). A semigroup only requires one algebraic property:

Associative:

$$(a \otimes b) \otimes c = a \otimes (b \otimes c) \ \ \forall a, b, c \in S$$

A semigroup may optionally have further properties, and may contain elements with special properties:

Commutative:

$$a \otimes b = b \otimes a \ \ \forall a, b \in S$$

Selective:

$$a \otimes b = a \ \vee \ a \otimes b = b \ \ \forall a, b \in S$$

Identity element $\alpha \in S$:

$$\alpha \otimes s = s \otimes \alpha = s \ \ \forall s \in S$$

Annihilator element $\omega \in S$:

$$\omega \otimes s = s \otimes \omega = \omega \ \ \forall s \in S$$

[1]For convenience we differ slightly from the usual definition of $\wp(T)$ as power set, which includes infinite subsets of T when T is itself infinite.

Examples of semigroups include $(\mathbb{N}^\infty, +)$ which has identity 0 and annihilator ∞; and $(\mathbb{N}^\infty, \min)$ with identity ∞ and annihilator 0. Both are commutative, but only $(\mathbb{N}^\infty, \min)$ is selective. The semigroup $(\wp(\mathbb{N}), \cup)$ has identity $\emptyset$ (the empty set) but no annihilator.

2.2.3 Bisemigroups

The shortest paths problem can be generalised using the algebraic structure of *bisemigroups*,

$$(S, \oplus, \otimes),$$

where the additive component $(S, \oplus)$ and the multiplicative component $(S, \otimes)$ are both semigroups. A bisemigroup may optionally have various properties:

Distributive:

$$a \oplus (b \otimes c) = (a \otimes b) \oplus (a \otimes c) \;\; \forall a, b, c \in S$$

Non-decreasing:

$$a = a \oplus (b \otimes a) \;\; \forall a, b \in S$$

Increasing:

$$a \neq \alpha_\oplus \Rightarrow a = a \oplus (b \otimes a) \neq (b \otimes a) \;\; \forall a, b \in S$$

Some of these properties are linked – any algebra that is increasing will also be non-decreasing, and typically only selective algebras can be non-decreasing.

For example, $(\mathbb{N}^\infty, \min, +)$ is a distributive, non-decreasing bisemigroup. From this, the generalisation of the shortest paths problem is straightforward: the path weight and edge weight type $\mathbb{N}^\infty$ is replaced by S, the weight concatenation operator $+$ is replaced by $\otimes$, and the best path selection operator min is replaced by $\oplus$. The constant 0 is replaced by the multiplicative identity $\alpha_\otimes$, while ∞ is replaced by the additive identity $\alpha_\oplus$. The Bellman-Ford algorithm (and similarly the distributed Bellman-Ford algorithm) can be generalised to work over bisemigroups by making the same replacements.

The special case of a distributive bisemigroup with a commutative $\oplus$, a multiplicative identity $\alpha_\otimes$, and an additive identity $\alpha_\oplus$ equal to multiplicative annihilator $\omega_\otimes$, is commonly known as a *semiring*, and its use in the generalised shortest paths problem has been explored extensively [Car79, GM84, GM08]. More recently it has become apparent that *non-distributive* bisemigroups are of importance for understanding Internet routing, leading to an exploration of their behaviour [Sob03, GG08, Gur09]. One important result is that while algorithms such as Bellman-Ford and matrix iteration will only compute a *globally optimal* solution (the chosen path between any two nodes will

be the best of all possible paths between them) if the algebra is distributive, they will still compute a stable solution that we call *locally optimal* with non-distributive algebras (given that the algebra is at least increasing). A node may not select the best of all paths in the network, but it will select the best of the paths that are an extension of its neighbour's selected paths. This is a valuable concept in network routing as it matches the common next-hop forwarding model, as well as matching the behaviour of BGP. The ability to relax the distributivity requirements provides far greater expressivity when designing routing algebras.

2.2.4 Preorders

Another important component of routing algebras is the *preorder*,

$$(S, \preceq)$$

where $\preceq$ is a binary relation over S. A preorder requires two properties:

Reflexive:

$$a \preceq a \ \ \forall a \in S$$

Transitive:

$$a \preceq b \wedge b \preceq c \Rightarrow a \preceq c \ \ \forall a, b, c \in S$$

Note that it is possible for values to be equivalent but inequal:

$$a \sim b \equiv a \preceq b \wedge b \preceq a.$$

Similarly it is possible for values to be incomparable:

$$a \# b \equiv \neg(a \preceq b) \wedge \neg(b \preceq a).$$

A partial order is an antisymmetric preorder (no inequal values are equivalent), and a total order is a partial order in which no values are incomparable:

Antisymmetric:

$$a \preceq b \wedge b \preceq a \Rightarrow a = b \ \ \forall a, b \in S$$

Total:

$$a \preceq b \vee b \preceq a \ \ \forall a, b \in S$$

A strict partial order $\prec$ (an irreflexive, transitive relation) can be derived from any preorder $\preceq$:

$$a \prec b \Leftrightarrow a \preceq b \wedge \neg(b \preceq a)$$

2.2.5 Order transforms

An alternative to bisemigroups in routing is *order transforms*: algebraic structures of the form

$$(S,\ L,\ \preceq,\ \triangleright)$$

where S and L are types, $\preceq$ is a preorder over S, and $\triangleright$ is a binary operator of type $L \times S \to S$. To apply order transforms to the shortest path problem, we use S for path weights, L for edge weights, $\preceq$ for picking the best of several path weights (we will assume it is a total order so there is always a single best), and $\triangleright$ for computing the weight of a path extended by an extra edge.

This is the structure we will use throughout most of this dissertation, as it provides the most appropriate match for the behaviour of complex vector routing protocols. As we move further away from the original model of shortest paths, we use different terminology: instead of weights we will call S the type of *metrics* (representing a potentially complex set of data about the path), L the type of *labels* or *policies* (representing a potentially complex set of decisions about routing behaviour associated with a network link), $\preceq$ encodes some notion of 'best' paths (not necessarily 'shortest' in any sense), and $\triangleright$ is the *policy application* function.

In an order transform, the preorder $\preceq$ is the counterpart to the bisemigroup's additive operator $\oplus$; there is a direct correspondence between the common cases of a total preorder and a selective $\oplus$ (typically defined as $a \preceq b \Leftrightarrow a = a \oplus b$). The application function $\triangleright$ is an asymmetric generalisation of $\otimes$ where the operator's two arguments are now of different types.

The algebraic properties that apply to bisemigroups correspond to properties over order transforms:

Distributive:

$$a \preceq b \Rightarrow c \triangleright a \preceq c \triangleright b \ \ \forall a, b \in S, c \in L$$

Non-decreasing:

$$a \preceq c \triangleright a \ \ \forall a \in S, c \in L$$

Increasing:

$$a \neq \omega \Rightarrow a \prec c \triangleright a \ \ \forall a \in S, c \in L$$

Infinity element $\omega \in S$:

$$c \triangleright \omega = \omega \ \ \forall c \in L$$

Distributivity can be thought of as 'order-preserving': if one path is preferred over another, then applying the same policy to both will not change the order of preference.

This is important in some situations as it means we can identify the 'best' path either before or after applying policy, and will get the same answer in either case.

There are many other similar algebraic structures that can be used. Order transforms can be expressed as $(S,\ \preceq,\ F)$ where arcs are labelled with functions from a set $F \subseteq S \to S$ and the path weight computation is $f(s)$ instead of $l \triangleright s$, but we avoid using this formulation as it maps less clearly onto implementations where configuration data is explicitly separated from computation. The quadrants model described by Griffin and Gurney [GG08] also includes semigroup transforms $(S,\ L,\ \oplus,\ \triangleright)$ and order semigroups $(S,\ \preceq, \otimes)$, while Sobrinho [Sob05] included a function mapping S onto a new 'weight' type W and defined $\preceq$ over W instead of over S – but the principles remain the same in each case.

2.2.6 Example algebras

There are a number of basic routing algebras which we will build on later. We have mentioned $(\mathbb{N}^\infty,\ \min,\ +)$ as a bisemigroup that can be used to implement the shortest paths problem. We can define a shortest-paths order transform algebra that works similarly:

$$sp = (\mathbb{N}^\infty,\ \mathbb{N}^\infty,\ \leq,\ +)$$

where $\leq$ is the standard integer less-than-or-equal order. This algebra is distributive and non-decreasing:

$$a \leq b \Rightarrow c + a \leq c + b \ \ \forall a, b, c \in \mathbb{N}^\infty,$$

$$a \leq c + a \ \ \forall a, c \in \mathbb{N}^\infty.$$

However, it is not increasing because we include 0 in L: the case $a = 0$ and $c = 0$ is a counter-example to the property

$$a \neq \infty \Rightarrow a < 0 + a \ \ \forall a \in S.$$

Some algebras can be best expressed by a finite table. The customer–provider–peer relationships and constraints described by Gao and Rexford [GR01] can be modelled algebraically [Sob03, GS05] with

$$cpp = (\{C,\ R,\ P,\ \infty\},\ \{c,\ r,\ p\},\ \preceq,\ \triangleright_{cpp}),$$

where $\preceq$ is defined by the order $C \prec R \prec P \prec \infty$. Figure 2.5 shows a simple hierarchical network labelled with this algebra. $\triangleright_{cpp}$ is defined by the table

$\triangleright_{cpp}$	C	R	P	∞
c	C	∞	∞	∞
r	R	∞	∞	∞
p	P	P	P	∞

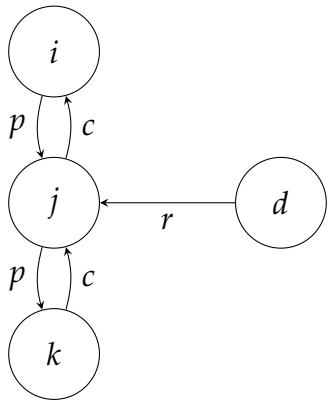


Figure 2.5: A customer–provider–peer network.

For example, say a route received by j from a peer has metric R. The arc to i from its customer j has label c. Since $c \triangleright_{cpp} R = \infty$, the route will be filtered out. Conversely, the arc to k from its provider j has label p. Since $p \triangleright_{cpp} R = P$, the same route sent across the arc from provider to customer will be accepted and given new metric P to indicate it came from a provider. This definition of $\triangleright_{cpp}$ guarantees that paths will satisfy the valley-free property defined by Gao [Gao00]: a route that travels 'down' the hierarchy cannot subsequently travel back up.

The *cpp* algebra is distributive and non-decreasing, which can be verified by enumerating all possible cases. Other algebras constructed in a similar manner may have different properties. Our table only included three rows (the policies c, r and p); Griffin [Gri10] explores all 24 possible rows (for a set of three metrics plus infinity) that retain the non-decreasing property but do not all retain distributivity.

We can also represent the path component of a path-vector routing protocol as a routing algebra

$$paths = (\mathrm{id}^*_{simp} \cup \{\infty\},\ \mathrm{id} \times \mathrm{id},\ \preceq_{listLen},\ \triangleright_{paths}).$$

The metrics are either lists of router identifiers with no repeated values, or are infinity. These identifiers are likely to be implemented as integers, but they can be any arbitrary data type as far as our language definition is concerned – we will refer to them as the type 'id'. Shorter lists are preferred over longer lists by $\preceq_{listLen}$, and infinity is least preferred. The arc from node i to node j is labelled with a *pair* of node identifiers, (i, j). We define $\triangleright_{paths}$ as

$$(i, j) \triangleright_{paths} l = \begin{cases} i :: l & \text{if } j \notin i :: l \\ \infty & \text{otherwise} \end{cases}$$

where :: means list prepending. In this definition, i is prepended to the path when it sends the route to j; but if j is already in this list, the route is filtered out (its metric is set to ∞) because this indicates a loop. This models the behaviour of BGP's `AS_PATH` attribute.

The *paths* algebra is increasing: $\triangleright_{paths}$ will either increment the length of the path or set it to infinity, making it less preferred under $\preceq_{listLen}$. It is not distributive: the metric $[1,2]$ is preferred over the metric $[1,3,4]$, but applying the label $(0,2)$ to both will result in the metrics ∞ and $[0,1,3,4]$, flipping the order of preference and violating distributivity.

2.2.7 Constructing algebras

Although it is possible to define algebras as in the previous section, and prove their algebraic properties by hand, this is slow and error-prone as the complexity increases. Metarouting is based around the concept of algebraic *constructors*: a routing language can be designed by breaking the problem down into simple components, each corresponding to a pre-defined algebra such as *sp* or *paths*, then using a selection of constructors to build up the components into the completed algebra.

One of the most important is the *lexicographic product* operator $\vec{\times}$. As an example, the algebra

$$sp \vec{\times} paths$$

has metrics that are pairs of values, one from *sp*'s metric set and one from *paths*'s; labels are similarly pairs; and the new algebra's $\triangleright$ applies labels to metrics componentwise based on $\triangleright_{sp}$ and $\triangleright_{paths}$. The new algebra's $\preceq$ is a lexicographic order: given two metrics $a = (a_1, a_2)$ and $b = (b_1, b_2)$, the new order is

$$a \preceq b \Leftrightarrow a_1 \prec b_1 \vee (a_1 \sim b_1 \wedge a_2 \preceq b_2).$$

In this example, the new algebra can be found to be increasing but non-distributive. Gurney and Griffin [GG07] show that these properties can be determined trivially for any lexicographic product algebra, when certain properties are known for the two component algebras.

The *direct product* operator $\times$ is very similar to $\vec{\times}$ but uses the non-lexicographic order

$$a \preceq b \Leftrightarrow a_1 \preceq b_1 \wedge a_2 \preceq b_2.$$

The *function union* operator $\uplus$ combines two algebras that have the same metric type:

$$(S,\ L_1,\ \preceq,\ \triangleright_1) \uplus (S,\ L_2,\ \preceq,\ \triangleright_2) = (S,\ L_1 \uplus L_2,\ \preceq,\ \triangleright_{\uplus}).$$

This performs a disjoint union of the label types; we distinguish values from the two sets by tagging them as 'in left' or 'in right'. The new operator $\triangleright_{\uplus}$ depends on the label tag:

$$\begin{aligned} \mathrm{inl}(l_1)\ \triangleright_{\uplus}\ s &= l_1\ \triangleright_1\ s \\ \mathrm{inr}(l_2)\ \triangleright_{\uplus}\ s &= l_2\ \triangleright_2\ s \end{aligned}$$

2.3 Vector routing protocols

Vector routing protocols are network protocols based on the distributed Bellman-Ford algorithm. They define the low-level behaviour of the routers in a network – the byte layout of network packets they transmit between each other, and the required computation (state machines for setting up sessions between routers, timing requirements to ensure the protocol stays responsive without overloading the network, route processing requirements to prevent routing loops, and so on) – as well as the intended interpretation of fields in the route metrics to be used in routing policy. The protocols implement the DBF algorithm as a (usually non-explicit and non-obvious) consequence of their requirements on network communication and processing.

Vector protocols are typically split into two categories based on their metric type: *distance vector protocols* such as RIP and EIGRP have metrics that represent only the cost of reaching the destination, whereas *path vector protocols* such as BGP use the metric to store some representation of the path that the route follows. Path vector protocols can use the path information to detect and prevent routing loops, whereas distance vector protocols need different mechanisms.

This section introduces the terminology and concepts used the RIP and BGP protocol specifications, as the rest of the dissertation will examine and extend these protocols. In particular, we describe them here from the low-level perspective in which the protocols are specified, and in Chapter 4 we recast them in the algebraic routing model.

A common concept in Internet routing protocols is the *destination prefix* [FL06]. This can usually be treated as an opaque identifier for the *source* node d in the shortest paths problems – that is, the source of routing information, but the destination of data traffic flowing back along the computed routes. (In this dissertation we focus almost exclusively on the flow of routing control messages, not the flow of data, so the directed edges in our graphs correspond to the direction in which routers will advertise information about the availability of a path to a destination prefix.)

Destinations on the Internet each have an IP address; for simplicity we will assume IPv4, in which an address is a 32-bit number expressed as a sequence of four 8-bit numbers such as “128.232.0.20”. There are often many separately-addressable destinations on a local network sharing a single connection to the Internet. To reduce the amount of information that Internet routers must maintain, destinations are grouped by a shared address prefix: “128.232.0.0/16” is a prefix representing the addresses whose first 16 bits are the same as the first 16 bits of 128.232.0.0, i.e. the range of addresses from 128.232.0.0 up to 128.232.255.255. These destination prefixes are used as the row indexes in the $\mathbf{X}$ matrix of the Bellman-Ford algorithm – the router will compute the distance to each prefix independently.

When a router receives a data packet, it must consider all routes whose destination pre-

fix matches the destination address of the packet. The address may match many prefixes, so *longest prefix matching* is used: if the router has routes for destination prefixes 128.232.0.0/16 and 128.232.0.0/24, both matching a destination address 128.232.0.20, it will pick the route 128.232.0.0/24 with the longer (hence more specific) prefix.

The functionality of a router can be considered as two separate layers. The *control plane* runs the routing protocol and computes the best path to each destination prefix, storing the result in a *routing table* (effectively a lookup table from destination prefix to metric and an identifier of the next hop on the path with that metric). The *forwarding plane* receives data packets and determines the next router in the path to send them on to, using longest prefix matching, based on data in a *forwarding table* (a lookup table from prefix to next hop). In practice the relationship between the routing tables and forwarding tables may be quite complex, especially in the cases of route redistribution between multiple protocols on a single router [LXZ07] and VPNs (virtual private networks) with multiple forwarding tables per router [RR06, BG03]. For the purposes of this dissertation we will treat the forwarding table as a straightforward mapping of the routing table, as we are working entirely in the control plane.

2.3.1 RIP

RIP (first specified in RFC 1058 [Hen88]; most recently in RFC 2453 [Mal98]) is a simple distance vector protocol designed for small networks. It is based on a "soft state" mechanism [Cla88]: there is no session setup handshake between neighbouring routers, and each router periodically advertises its best known distance to each destination prefix, so a router that is added to the network or recovering from an error can simply wait for a short time period (typically 30 seconds) to receive all route advertisements and reconstruct its routing table. Instead of remembering the routes advertised by each neighbour, the router simply remembers the single best route in its routing table, and updates it immediately when receiving an advertisement message that either has a better metric than the current best route, or is sent by the router that provided the current best route. This is an asynchronous implementation of the distributed Bellman-Ford algorithm, with each router updating its column of the matrix $\mathbf{X}$ based on its neighbours' advertisements of their own columns of $\mathbf{X}$, so it will iteratively converge on a correct routing solution.

Route metrics in RIP are simply integers from 1 to 15. The cost associated with a link is a positive integer, typically 1. This means the maximum diameter of the network (i.e. the longest of all shortest paths) is 15 hops – if the network is too large then some routers will be unable to find paths to each other.

Routers running RIP communicate using UDP with no reliability mechanism; packet loss will result in route advertisement messages being missed until they are readver-

tised 30 seconds later. To cope with persistent network failure or routers being removed from the network, a router that has received a route advertisement will expire it after some time period (typically 180 seconds) if it has not been readvertised.

RIP deals with the *counting to infinity* problem of the DBF algorithm through a number of means. Firstly, the maximum metric is a small integer (15) so "infinity" can be counted to reasonably quickly without a great waste of network resources. Secondly, *triggered updates* aim to make the protocol count to infinity faster: whenever a router makes a change to its routing table, it should advertise its new routes almost immediately (typically within 1 to 5 seconds; the small delay is to avoid overloading the network) instead of waiting for the 30 second timer.

Thirdly, the *split horizon* feature specifies that if some router a learned its best route to some destination from router b, it must not advertise that route back to b. A route $[\ldots, b, a, b, \ldots]$ could not possibly be a shortest path (it would be shorter to remove the cycle) so omitting it does not affect the correctness of the routing solution, and it prevents counting to infinity around a cycle between two adjacent nodes (though it has no effect on counting around longer cycles). Finally, the *poisoned reverse* feature is a modification of split horizon: if a and b each think the best route is through each other (due to dynamic network changes creating a temporary loop), split horizon would result in neither of them advertising the route to the other, and each would retain that route until a timeout is reached. Poisoned reverse allows them advertise the route with the metric replaced by an "infinity" marker, causing the opposite router to immediately stop using that route, leading to faster convergence of the protocol.

All these features are a tradeoff between convergence time and network traffic overhead and protocol complexity, trying either to prevent the counting to infinity problem or to make it terminate sooner.

2.3.2 BGP

Whereas RIP is designed for use in small networks, BGP [RLH06] is designed to connect the whole Internet together (albeit a smaller Internet than today's, leading to some scalability problems). It does this with a two-level hierarchy based on *autonomous systems* (ASes), illustrated in Figure 2.6. An AS is a group of routers – sometimes a single router, sometimes thousands spread across the world – identified by a 16-bit[2] number. Each router can only belong to a single AS.

Within an AS, BGP assumes full mesh connectivity (every router is connected directly to every other router). This could be implemented with physical links between every

[2] A scalability problem as the number of registered ASes has now exceeded 2^{16}, requiring some hacks to expand to a 32-bit number without breaking compatibility [VC07]; we will ignore this detail in the rest of this dissertation and assume 16-bit AS numbers.

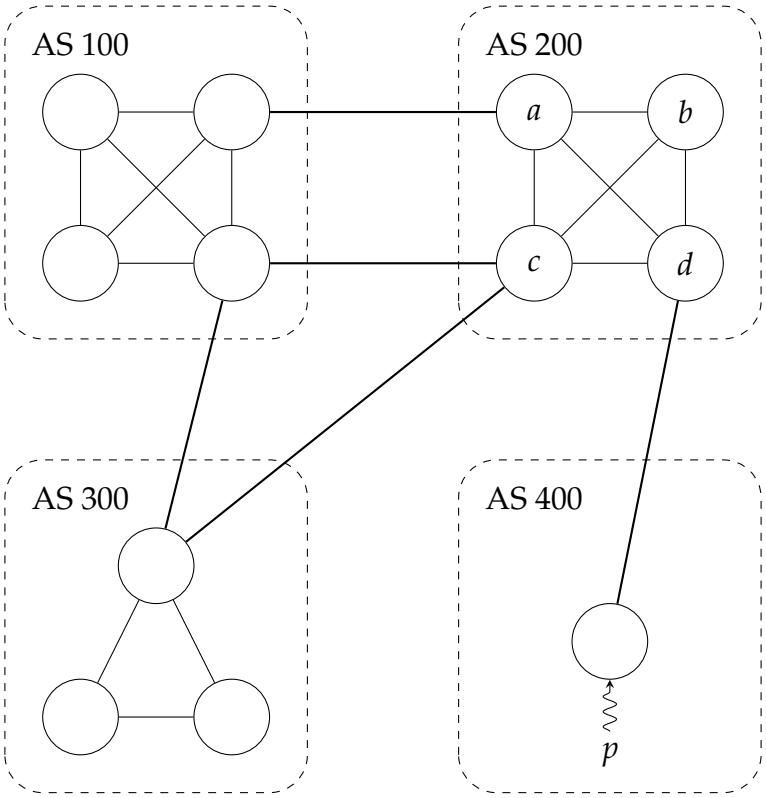


Figure 2.6: Autonomous systems connected by BGP.

pair of routers, but in practice the connectivity is typically provided by a separate routing protocol such as RIP or OSPF running inside the AS, referred to as the AS's Interior Gateway Protocol (IGP). The IGP computes paths between each pair of routers over a non-mesh physical network, and BGP implements its mesh on top of this layer without needing to know the details of the physical network layout.[3]

In contrast to RIP, BGP is a "hard state" protocol: routers communicate using sessions over reliable TCP channels, with a complex state machine to run the session handshake and subsequent communication. There are many optional extensions to BGP, and this handshake allows routers to exchange a list of their capabilities and determine which features can be safely used. When two routers set up a session, they first send their whole routing table (subject to filtering) to each other, then subsequently only send data for changes in their routing table. This allows the communication cost to scale with the number of route changes in the network, rather than the total number of routes.

BGP routers in the same AS (which is detected in the session handshake process) communicate in Internal BGP mode (IBGP). The full mesh makes this relatively straight-

[3] BGP route reflectors [BCC06] provide an alternative to mesh connectivity, improving scalability in larger networks, but are designed to provide similar behaviour to the mesh and do not significantly change the operation of BGP, so will ignore this detail too.

forward. When router d in Figure 2.6 learns of a route to destination address prefix p from the router outside its AS, it advertises the new route directly to every other router in its AS. When router c learns that route from d, which it knows is inside its AS, it does not need to advertise it to a or b as they will have already received it directly from d. This means a route will never traverse more than a single link inside the AS, preventing intra-AS routing loops.

BGP routers that are connected but that are in different ASes communicate in External BGP mode (EBGP). There are too many ASes for a full mesh to be feasible, so EBGP relies on the distributed Bellman-Ford algorithm, with an AS acting collectively as a single node: each router advertises its known routes to the neighbouring routers in different ASes, while IBGP keeps the router synchronised with the others in its own AS. Although IBGP cannot form loops inside a single AS, EBGP can form loops through a cycle of ASes. To detect and eliminate looping routes, BGP stores each route with an *AS path*, and a router adds its own AS number onto the front of the path when advertising the route over EBGP. For example, a router in AS 300 may see a route to p with path "100 200 400". If this route was advertised back to router c in AS 200, the router would detect that 200 is already in the list and reject the route as it would form a cycle.

The AS path is also used as a part of the route's metric, when a router picks the best of multiple available routes to a destination; shorter paths are preferred over longer paths, all else being equal. The metric contains a number of other *BGP attributes*, each of which has a name (specified in the BGP RFC or in extension RFCs, to ensure global agreement on the semantics of each name) and a value (sometimes an integer, sometimes a more complex structure such as a list; the attribute name determines the value's data type). The BGP *decision process* for picking the best route is based on examining the attributes in sequence and rejecting routes that are not the best in that attribute, until a single route remains. The sequence is defined as:

1. `LOCAL_PREF` (local preference) – this attribute is set by an AS's router when receiving routes from EBGP (i.e. from outside the AS), giving the AS full control over route selection within itself. 32-bit integer; higher values are preferred.

2. `AS_PATH` – shorter paths are preferred. Because local preference is checked before path length, BGP is not a shortest paths algorithm, but length is checked before any other attributes to discourage the selection of very long paths. A router prepends its AS number to the front of the list when sending the route to another AS over EBGP.

3. `ORIGIN` – set when the route is originated and never changed by other routers. It has the value "IGP" (the route was originated by the BGP router itself, based on no external information), "EGP" (learned from an External Gateway Protocol that is not BGP; no such EGP is used nowadays so this value is mostly obsolete), or

"incomplete" (learned from some other source, e.g. through route redistribution from another protocol on the same router). "IGP" is preferred over "EGP", which is preferred over "incomplete".

4. `MULTI_EXIT_DISC` (multiple exit discriminator, or MED) – set by an AS's router when sending the route over EBGP to another AS. If the first router has multiple connections (exits) to the second AS, this lets it influence which one the second AS will prefer to use. 32-bit integer; lower values are preferred, but by default two MED values are considered incomparable if they were set by different ASes (effectively this is a partial order and each AS has its own set of MED values disconnected from all others; it would be meaningless to compare MEDs from different ASes as integers).

5. Prefer EBGP over IBGP – provides a form of "hot potato" behaviour by preferring routes that leave the current AS (avoid the cost of transiting traffic through the AS).

6. `IGP_COST` – if the route was received over an IBGP connection, this is the cost of the IGP path underlying that IBGP link (typically the integer metric from RIP or OSPF or EIGRP). Routes with a lower cost are preferred, so the AS will choose the route that is least expensive to transit over its own network (subject to all earlier constraints).

7. Advertiser's BGP Identifier – if a single best route has still not been selected, the 'identifier' of the BGP router that advertised the route (typically an IP address of a network interface on that router, which ought to be globally unique) is used as an arbitrary tie-breaker.

8. Advertiser's peer address – poorly-configured networks may have multiple routers with the same BGP identifier. As a last resort, a selection is made based on the IP address that the route was received from, which is necessarily unique since a router cannot have multiple connections to a single IP address.

BGP *policy* is the set of rules configured by a network administrator that determine how to alter these attributes when receiving routes or advertising routes to neighbouring routes. The BGP RFC does not specify how policy should be configured or how much flexibility should be provided; it just defines the semantics of the attributes and a few constraints on their usage, leaving the rest as an implementation detail.

MED details

The standard definition of MED comparison has led to a number of problems, and is a useful demonstration of the difficulties caused by BGP's approach to routing protocol

design. It has been shown [MGWR02, GW02] that certain uses of the MED attribute can trigger persistent oscillations, preventing BGP from ever converging to a stable solution and wasting network resources. Even when these problems are avoided by careful network configuration, some implementations of the BGP decision process can result in incorrect path selection that is not determined solely by the route metrics. For example, Cisco's original BGP implementation worked by iterating over the list of routes in an arbitrary sequence (depending on the timings of when route advertisements were received), maintaining a single pointer to the best route seen so far, and updating it whenever the iteration reaches a better route. Consider the following sequence of routes:

	AS_PATH	MED	IGP_COST
A	100 ...	10	5
B	200 ...	10	4
C	100 ...	20	3

(Omitted attributes are assumed to be identical for all routes.) *A* and *B* were received from different ASes, so their MEDs are incomparable. Instead the IGP cost is compared, and *B* is better than *A*. Now the best route, *B*, is compared against *C*, and *C* has a lower IGP cost so it is selected as the best. However, if *A* and *C* were compared to each other then *A* would be preferred as they were received from the same AS (so their MEDs are comparable) and *A* has a lower (better) MED value. The order of comparisons has resulted in a route being selected despite there being a better route available.

This happens only because of the special behaviour of MED; the sequential algorithm is correct except for this detail. Cisco added the configuration option named "`bgp deterministic-med`" [Cis09b] to fix this behaviour (by sorting routes by AS path before iterating over them), and documentation for network administrators "strongly recommends" this option [GS02], but it is disabled by default to preserve backward compatibility with networks that were designed with the buggy functionality. (Juniper's implementation of BGP even has a "`cisco-non-deterministic`" option [Jun10b] to *enable* this behaviour, for compatibility.) It is unclear whether the Cisco developers were aware of this problem when first implementing BGP – the deterministic-med command was not present in early versions, suggesting it was not seen as a problem in need of fixing, but it has been anecdotally claimed [Whe03, MG06] that the incorrect MED handling was an intentional divergence from the RFC to ignore all but the most stable routes and prevent rapidly-changing MED values from overloading routers.

In any case, the MED attribute introduces a disproportionate amount of complexity into the analysis and safe use of BGP. It is an important example of the two approaches to understanding routing protocols that this dissertation makes use of. The BGP RFC glosses over all these complexities of MED, simply describing the mechanical processing of the attribute and leaving the analysis of its behaviour and its implementation

difficulties and the necessary guidelines for safe usage as a task for later work (by RFC writers, academics, implementers, or network administrators). In contrast, algebraic routing demands that these problems are confronted in advance: before a feature like MED could be included in a routing algebra, it must be understood well enough to prove its usage is safe, and hence not included if it is impossible to prove it is safe. Whether this is an advantage or disadvantage depends on one's point of view: traditional protocol design has been willing to compromise safety for functionality, whereas metarouting aims to improve on the state of the art by viewing safety as fundamental and inviolable at the expense of some flexibility. A goal of this dissertation is to help reconcile these models by demonstrating that important functionality can be included without sacrificing safety.

Communities

BGP is designed to be extensible, and many extensions have been designed and implemented. One of the most important is the *communities* attribute [CTL96]. This is an optional attribute that is added to the route metric, containing an unordered list of community values. A community value is a pair of 16-bit integers: the first is an AS number, and the second is a value whose meaning is defined by the administrator of that AS. This numbering system is designed to provide globally unique identifiers without a centralised registry.

The BGP decision process is not affected directly by communities, but policies can identify routes containing particular communities and then update other attributes. For example, the administrator of AS 100 might set up its policy configuration so that if it receives a route with community value $(100, 10150)$ then it will set `LOCAL_PREF` to 150, and a route with $(100, 10160)$ will have `LOCAL_PREF` set to 160. AS 200 might then choose to set up policy so that router a (from Figure 2.6) adds $(100, 10150)$ to the community attribute of routes that it is advertising to AS 100, while router b adds $(100, 10160)$ instead. This gives AS 200 some power to influence the decision process inside AS 100 (in this case making routes through b preferred over any routes through a, similar to use of MED but without the associated problems) while letting AS 100 keep ultimate control over how it interprets those community values.

Donnet and Bonaventure [DB08] found around 7000 distinct community values used by around 1000 distinct ASes in published routing table dumps from 2007. They also provide a taxonomy of publicly-documented community values. Although communities may be interpreted in arbitrary ways by routing policy, the vast majority fall into a small number of classes: setting `LOCAL_PREF`; tagging where a route has been received from (geographic location, AS number, etc.); controlling whether a route should be advertised onwards to particular groups of routers (identified by geographic location, AS number, etc.); and controlling AS path prepending (the AS that receives the

route will prepend its AS number more than once onto the `AS_PATH` attribute, making the path longer (less preferred) without affecting the cycle-detection behaviour of the attribute).

2.4 Protocol implementations

There are several major implementations of the RIP and BGP routing protocols. We will explore some of their details later, to help with reconciling the theoretical and practical perspectives on routing. This section gives a brief overview to put them into context, and describes some details of the Quagga implementation that we build our metarouting prototype system on top of.

Much of the routing market has been dominated by Cisco. Most of their routers run the Cisco IOS software, which implements many routing protocols (including RIP and BGP) along with a great deal of other functionality. IOS was originally designed as an embedded system in the 1980s, running in a single address space with no memory protection and with a non-preemptive scheduler [BWM08]. The software architecture has been improved incrementally over time [Cis06], but without changing the basic kernel design. Backward compatibility has been an important part of the evolution of IOS: router configuration files written for early versions are supported by new versions of IOS and have the same semantics, as far as possible, even when this results in sub-optimal behaviour (such as the "`bgp deterministic-med`" command mentioned earlier, which is strongly recommended but still not enabled by default).

More recently, higher-end products run Cisco IOS XR, a redesigned and largely rewritten-from-scratch alternative to IOS based on a real-time operating system microkernel (QNX) with more modern features allowing improved reliability. The development of IOS XR also provided an opportunity to break backward compatibility: the configuration file syntax remained broadly similar but changed in a number of details (for example the `deterministic-med` mode was made the default and the option was removed, and the language for writing routing policy was completely changed).

The second major commercial router vendor is Juniper. Their software, JunOS, is based on the FreeBSD operating system and has a modular architecture that is closer to IOS XR than to IOS [Ker08]. Its configuration file syntax is superficially very different to Cisco's and has a more uniform structure, though it exposes similar functionality.

While the commercial routers provide custom hardware for their software, there are also a number of open source software routers that work on standard PC hardware running (typically) Linux or BSDs, making them much more useful for research and for low-cost deployments. Quagga is one of the more widely used implementations; it is hard to measure usage with any reliability, but an incident that caused most Internet-

connected Quagga routers to simultaneously die[4] caused about 2500 destination address prefixes (roughly 1% of the global total) to vanish [Zmi09], indicating that it has a small but non-negligible usage share. Quagga's configuration file syntax is based heavily on Cisco IOS, though the implementation is very different.

Quagga has a modular architecture, with each routing protocol (BGP, RIP, OSPF, etc.) running as a daemon in a separate Unix process. Another process, named "zebra", provides an interface between the routing protocol daemons and the operating system's forwarding tables, and between daemons that need to communicate with each other. All communication between processes is performed with sockets (either local Unix sockets if the processes are running on the same machine, or TCP sockets if they are spread over multiple machines). There is also a shared library of common functionality (configuration parsing code, hash tables, prefix lists, etc.) that all the processes can use. Quagga is written in C. The coding style avoids complex abstractions, so the protocol implementations can be verbose and contain a lot of boilerplate code but are fairly straightforward to understand and to modify.

XORP [HKG+05] is a more recent open source routing implementation, with a very different approach to Quagga. XORP breaks down the protocols into smaller modules, with the aim of providing extensibility to support protocol research. It is written in C++ with some complex abstractions (including dynamically-modifiable pipelines, event-driven interfaces, and a custom extensible IPC mechanism) and well over twice as many lines of code as Quagga for similar functionality. Its configuration syntax is based roughly on Juniper's.

[4] Its handling of 32-bit AS numbers (mentioned in an earlier footnote) was buggy – the code was not compatible with numbers longer than 5 digits. The bug was ironically triggered by a network starting to use 32-bit AS numbers in an attempt to demonstrate that they were safe to use.

Chapter 3

Language definitions

As introduced in Section 1.2 the metarouting system defines two programming languages, named RAML and ERL, to compile a high-level routing algebra specification into executable code that can run in the Quagga routing software. This chapter presents a detailed description of the semantics of the RAML and ERL languages, along with some illustrative examples. The full extent of the complete languages is growing over time and is beyond the scope of this work, so we focus on a subset: in particular we use RAML primarily to express algebras that are order transforms (whereas a complete version should support others such as bisemigroups), and only include the RAML and ERL expressions that are needed for the examples in this dissertation. However, we go into this subset in depth and fill out many of the details necessary for a practical implementation.

Metarouting is fundamentally a constructive system, and the languages reflect this. Their syntaxes encode tree structures with leaf nodes being pre-defined base expressions or identifiers and with non-leaf nodes being pre-defined constructor expressions. The semantics of any expression is determined by that expression plus the semantics of its sub-expressions.

The origin of RAML is the paper by Griffin and Sobrinho [GS05] introducing the goal of defining a common language capable of expressing a large range of useful routing algebras, from which the algebraic properties can be automatically derived in order to check correctness conditions. It suggests a number of base algebras (integer plus, integer min, lists of integers with prepending, etc.) and constructors (lexicographic product, scoped product, disjoint union) that this language should include, to model protocols similar to BGP. Gurney and Griffin [GG07] examine the lexicographic product constructor in more detail and show that the scoped product is made redundant by it, along with adding 'left' and 'right' constructors to support this use. Minimal sets have been added to model equal-cost multipath routing [GG08], and the addition of *semi-module* algebras and associated constructors has been proposed for modelling route redistribution and the distinction between routing and forwarding [BG09].

In this way, the design of RAML has been driven forwards by the desire to understand and implement the behaviour of current routing practices, rather than by an attempt to comprehensively implement algebras suggested by literature in other fields (such as the examples of Gondran and Minoux [GM08]); network routing is the focus of this language design, and algebras are a means to that end. In parallel with this drive for routing expressivity, the language design is restricted by the need to reason about the properties required by the algebras. Adding a new constructor to the language requires inference rules for its properties, and those rules may rely on new properties that must then be computed for every other constructor and base algebra in the language. These two forces require careful balance: the hope is that we can add enough expressivity to the language to be useful, while allowing it to be understood with a set of properties that remains finite. The most extensive work on this problem is by Naudžiūnas [Nau11], using Coq to help derive properties and rules for a large subset of the language, but it is not yet entirely clear whether this approach will continue to be successful or will hit intractable problems as we further increase expressivity. In this dissertation we focus on parts of the language that we believe satisfy this balance, but the evolution of the language is a complex and critical part of ongoing research.

Billings [Bil09] gives the first detailed description of an implementable version of the RAML language, and introduces the ERL language as an important part of the implementation. While it provides a good overview of the languages, that description has a few shortcomings. Foremost, it defines the algebraic semantics of ERL but not of RAML. To determine the algebraic properties of a RAML specification, it requires that the RAML syntax be translated into ERL syntax, then the ERL syntax translated into its algebraic semantics, and then the properties found by looking up the algebra in a table of algebras and properties. Both translations are well defined, but the final lookup is relying on complex pattern-matching and can provide no guarantee that the patterns cover all algebras derived from RAML specifications. (The patterns certainly cannot cover all ERL specifications, as ERL can express algebras whose properties cannot be determined – only RAML (which is less expressive) is designed to support full property inference.)

Further, this approach of translating to ERL before determining algebraic properties does not match our real implementation of the languages, and would introduce extra complexity into an implementation. For example, error messages triggered by incorrect algebraic properties would have to be translated from the ERL layer back into the RAML layer before being displayed to the user; it would be simpler and more reliable to generate them from the RAML directly.

Additionally, its definition includes non-primitive algebraic structures (bisemigroups, order transforms, etc.) as core parts of the ERL language. In this dissertation we will use a variety of larger structures (e.g. Chapter 5 splits policy labels into many separate types) and it is inconvenient to have to extend the definition of ERL for every one of

these.

Given the importance of RAML and ERL as foundational components of the metarouting system, and as a base that will be extended later in this dissertation, we use this chapter to redefine them without those shortcomings. As illustrated earlier in Figure 1.4, we can translate RAML syntax directly into its algebraic semantics (including algebraic properties) without going through ERL. This ensures the completeness of property inference and matches the real implementation. Our model of ERL no longer includes complex structures such as order transforms; instead it supports an arbitrary named collection of primitive structures (types, preorders, transforms, etc.). RAML still understands order transforms (as its property inference rules are defined in terms of those structures) but they are flattened into two types plus a preorder plus a transform during the translation to ERL. This allows us to reuse ERL for more complex structures without any changes to ERL itself.

This chapter's definition of ERL's syntax and semantics matches the current implementation of the ERL-to-C++ compiler, named *mrc*. The implementation of RAML is in flux at the time of writing, so the syntax described here is somewhat hypothetical in its details but matches the implementation effort in its architecture and its concepts. The design and implementation of ERL has been largely driven by John Billings, and RAML has been developed by various members of the metarouting group; the new contribution of this dissertation is the concrete syntax for ERL and the first accurate definition of the languages, and of their relationships with each other and with the routing algebras.

3.1 Introduction to ERL

ERL is the intermediate language used for declaratively encoding the data types and computational behaviour of a routing language. It is designed to be expressive enough for a wide range of routing languages, while still providing safety guarantees (e.g. not supporting computations that might hang or crash) and being feasible to analyse algebraically (so the correctness of a RAML-to-ERL compiler can be robustly verified).

3.1.1 Semantic domain

The semantic interpretation of an ERL program is a collection of named expressions, each of one of the following kinds:

- **Types**: sets of values.
- **Preorders**: relations over values of a given type, with reflexivity and transitivity.

- **Semigroups**: binary operators over values of a given type (returning a value of the same type), with associativity.
- **Constants**: values of a given type.
- **Transforms**: total functions that take a non-zero number of arguments of given types, and return a value of another given type.

For example, an ERL program to implement a simple shortest-paths routing language with 'infinity' values could have the following algebraic semantics:

$$
\begin{aligned}
\texttt{sig} &= \{\omega\} \cup \mathbb{N} \\
\texttt{lbl} &= \{\omega\} \cup \mathbb{N} \\
\texttt{ord} &= (\texttt{sig}, \preceq) \text{ where } u \preceq v \Leftrightarrow v = \omega \vee (u \neq \omega \wedge v \neq \omega \wedge u \leq v) \\
\texttt{tfm} &= \lambda ls. \begin{cases} \omega & \text{if } l = \omega \text{ or } s = \omega \\ l + s & \text{otherwise} \end{cases}
\end{aligned}
$$

where `sig` and `lbl` are types (non-negative integers plus the special value identified by "ω"); `ord` is a preorder over the type `sig` (any integer is less preferred than ω; otherwise integers are compared with $\leq$); and `tfm` is a transform with two arguments of types `lbl` and `sig` respectively, returning a value of type `sig`. (We use the standard lambda calculus syntax $\lambda a_1 a_2.e$ to represent functions with (in this case) two arguments, usually with implicit types.)

3.1.2 Syntax

The grammar of ERL is defined in Figures 3.1, 3.2 and 3.3. As ERL is primarily an intermediate language, the syntax is simple and verbose with type annotations for every sub-expression, to allow straightforward parsing. In the grammar we write "x, x, ..." as a shortcut for expressing a list of zero or more occurrences of x. The next section will introduce this syntax more gradually by example.

For convenience we also allow any expression to be the *name* of an earlier `let` statement of the same kind; these references are immediately expanded and replaced by the referenced expression during the parsing process, and have no further effect on the behaviour of the compilation process, so we omit this detail from the definition of syntax for simplicity.

3.1.3 ERL by example

As an example of this syntax, the shortest-paths ERL program introduced in Section 3.1.1 could be expressed as in Figure 3.4.

```
erl-ty-unit ::= TyUnit
erl-ty-bool ::= TyBool
erl-ty-int-unbounded ::= TyInt
                       | TyIntNonNeg
                       | TyIntPos
erl-ty-int ::= erl-ty-int-unbounded
             | TyIntRange(num, num)
erl-ty-string ::= TyString
erl-ty-addconst ::= TyAddConst(name, erl-ty)
erl-ty-list ::= TyList(erl-ty)
              | TyListSimp(erl-ty)
erl-ty-set ::= TySet(erl-ty)
             | TySetMin(erl-ty, erl-po)
erl-ty-record ::= TyRecord(name=erl-ty, name=erl-ty, ...)
erl-ty-union ::= TyUnion(name=erl-ty, name=erl-ty, ...)

erl-ty ::= erl-ty-unit
         | erl-ty-bool
         | erl-ty-int
         | erl-ty-string
         | erl-ty-addconst
         | erl-ty-list
         | erl-ty-set
         | erl-ty-record
         | erl-ty-union
```

Figure 3.1: Grammar for ERL types.

```
erl-po ::= PoIntLte : erl-ty-int
         | PoListLenLte : erl-ty-list
         | PoAddTop(erl-po) : erl-ty-addconst
         | PoAddBot(erl-po) : erl-ty-addconst
         | PoDual(erl-po) : erl-ty
         | PoEquiv : erl-ty
         | PoIncomp : erl-ty
         | PoRecord(name=erl-po, name=erl-po, ...) : erl-ty-record
         | PoRecordLex(name=erl-po, name=erl-po, ...) : erl-ty-record
         | PoUnion(name=erl-po, name=erl-po, ...) : erl-ty-union
         | PoUnionOrdered(name=erl-po, name=erl-po, ...) : erl-ty-union
```

Figure 3.2: Grammar for ERL preorders.

```
erl-sg ::= SgIntPlus : erl-ty-int-unbounded
         | SgIntRangePlus : TyAddConst(name, TyIntRange(num, num))
         | SgIntMin : erl-ty-int
         | SgIntMax : erl-ty-int
         | SgAddAlpha(erl-sg) : erl-ty-addconst
         | SgAddOmega(erl-sg) : erl-ty-addconst
         | SgLeft : erl-ty
         | SgRight : erl-ty

erl-binop ::= BoListCons
            | BoSemigroup(erl-sg)

erl-expr ::= name
           | ExprUnit : erl-ty-unit
           | ExprInt(num) : erl-ty-int
           | ExprString("str") : erl-ty-string
           | ExprList(erl-expr, erl-expr, ...) : erl-ty-list
           | ExprSet(erl-expr, erl-expr, ...) : erl-ty-set
           | ExprBinop(erl-binop, erl-expr, erl-expr) : erl-ty
           | ExprApply(erl-transform, erl-expr, erl-expr, ...) : erl-ty
           | ExprCond(erl-expr, erl-expr, erl-expr) : erl-ty
           | ExprRecord(name=erl-expr, name=erl-expr, ...) : erl-ty-record
           | ExprSelect(erl-expr, name) : erl-ty
           | ExprConstCase(erl-expr, name, erl-expr, erl-expr) : erl-ty
           | ExprInject(name, erl-expr) : erl-ty-union
           | ExprSwitch(erl-expr, name=erl-transform, name=erl-transform, ...) : erl-ty

erl-transform ::= fun (name:erl-ty name:erl-ty ...) -> erl-expr

erl-statement ::= let type name = erl-ty
                | let preorder name = erl-po
                | let semigroup name = erl-sg
                | let constant name = erl-expr
                | let transform name = erl-transform
erl-program ::= erl-statement
              | erl-statement erl-program
```

Figure 3.3: Grammar for ERL semigroups, transforms and programs.

```
let type sig = TyAddConst(W, TyIntNonNeg)

let type lbl = TyAddConst(W, TyIntNonNeg)

let preorder ord =
  PoAddTop(PoIntLte : TyIntNonNeg) : TyAddConst(W, TyIntNonNeg)

let transform tfm =
  fun (l:TyAddConst(W, TyIntNonNeg) s:TyAddConst(W, TyIntNonNeg))
    -> ExprBinop(
         BoSemigroup(
           SgAddOmega(SgIntPlus : TyIntNonNeg) : TyAddConst(W, TyIntNonNeg)
         ), l, s
       ) : TyAddConst(W, TyIntNonNeg)
```

Figure 3.4: Shortest-paths routing language expressed in ERL.

The simplest class of expressions in ERL is types. The first two statements in this example declare types named `sig` and `lbl`. The semantics of an ERL type expression is a set of values, for example the semantics of "`TyIntNonNeg`" is the set of all non-negative integers, $\mathbb{N}$. Some types are parametric: the semantics of "`TyIntRange(0, 15)`" is the set of integers in the range $[0, 15]$. We will express the semantics of ERL syntax with the $[\![\ldots]\!]$ operator:

$$\begin{aligned} [\![\texttt{TyIntNonNeg}]\!] &= \mathbb{N} \\ [\![\texttt{TyIntRange(}n\texttt{, }m\texttt{)}]\!] &= \{i \in \mathbb{Z} \mid n \leq i \leq m\} \end{aligned}$$

Some types are constructed from other types: "`TyAddConst(W, TyIntNonNeg)`" is the set $\mathbb{N}$ plus the constant identifier `W`. In general:

$$[\![\texttt{TyAddConst(}c\texttt{, }t\texttt{)}]\!] = \{c\} \cup [\![t]\!]$$

These constant identifiers can be arbitrary uppercase strings. (We use case to distinguish constants from references to expressions named by `let`, which must be lowercase.) The same string can be used for multiple identifiers when they are unambiguous; it is fine to use the string `W` to represent an infinity for many different types in the same ERL program, since the context can resolve any ambiguity. In this example, we use it to represent the infinity values for both `sig` and `lbl`.

Other ERL types listed in the earlier syntax include strings (opaque sequences of ASCII characters), lists (whose elements are all of the same type), simple lists (lists where no two elements are equal to each other), sets, and minimal sets (which have a type and a preorder, and enforce the condition that no element in the set is strictly greater (by that

preorder) than any other element). `TyUnit` is the unit type, $\{1\}$, which can take only a single value; it cannot carry any data by itself but is useful in special cases. Record and union types are discussed later.

ERL preorder expressions define relations over a type. Every expression is of the form "p `:` t", for example "`PoIntLte : TyIntNonNeg`" defines the integer-less-than-or-equal order over $\mathbb{N}$.

There are restrictions on t for each preorder expression: `PoIntLte` can be applied to any integer type (`TyIntRange`, `TyInt`, `TyIntPos`, `TyIntNonNeg`), but using it over any other type is an error. Similarly, `PoAddTop` requires a type of the form `TyAddConst(`c`,` t'`)`, and has the semantics that every value in t' is lower than the value c.

The example program uses "`PoIntLte : TyIntNonNeg`" as a sub-expression inside the `PoAddTop` constructor, with the resulting type `TyAddConst(W, TyIntNonNeg)`. We define the semantics as

$$\begin{aligned} [\![\texttt{PoIntLte : TyIntNonNeg}]\!] &= \preceq \\ [\![\texttt{PoAddTop(}p\texttt{:}t\texttt{) : TyAddConst(}c\texttt{, }t\texttt{)}]\!] &= \preceq \text{ where } s_1 \preceq s_2 \Leftrightarrow \\ & \quad s_2 = c \vee (s_1 \neq c \wedge s_2 \neq c \wedge s_1 \, [\![p]\!] \, s_2) \end{aligned}$$

The definition of `PoAddTop` says that any value is less than or equal to c; otherwise, if neither value is c (and therefore must be in t) it defers to the preorder on type t defined by p.

`PoAddBot` acts similarly, adding a value that is more preferred than any value in t. `PoEquiv` gives the trivial order in which every element is equivalent ($s_1 \preceq s_2$ for all s_1, s_2). `PoIncomp` gives the trivial order in which every element is incomparable ($s_1 \preceq s_2 \Leftrightarrow s_1 = s_2$). `PoDual` reverses an order:

$$[\![\texttt{PoDual(}p\texttt{:}t\texttt{) : }t\texttt{)}]\!] = \preceq \text{ where } s_1 \preceq s_2 \Leftrightarrow s_2 \, [\![p]\!] \, s_1$$

Semigroup expressions work similarly to preorders, defining a binary operator over a type. In the example, "`SgIntPlus : TyIntNonNeg`" defines the semigroup $(\mathbb{N}, +)$. `SgAddOmega` is analogous to `PoAddTop`, creating a semigroup where the constant value c acts as an annihilator ($c \otimes s = s \otimes c = c$ for all s). `SgAddAlpha` makes c act as an identity ($c \otimes s = s \otimes c = s$ for all s).

`SgIntPlus` can only be applied to unbounded integer types. Bounded integers require a way to handle results that are outside the bounds, so we define a special semigroup `SgIntRangePlus` which applies to types `TyAddConst(`c`,` `TyIntRange(`n`,` m`))` and will return the value c if the result would overflow. This lets ERL avoid any kind of run-time exception handling, as the language definition ensures at compile-time that any expression with any arguments will produce a valid result. An alternative semigroup could simply clamp the result to the valid range, but we have found that returning

a special constant is more useful in practice and have not added the clamped-plus semigroup to this version of ERL.

`SgIntMin` and `SgIntMax` return integer minimum/maximum values. `SgLeft` and `SgRight` always select one of their arguments:

$$\begin{aligned} [\![\texttt{SgLeft}]\!] &= \otimes \text{ where } s_1 \otimes s_2 = s_1 \\ [\![\texttt{SgRight}]\!] &= \otimes \text{ where } s_1 \otimes s_2 = s_2 \end{aligned}$$

The most complex class in ERL is transform expressions, which represent computations in a specialised functional language. The example defines `tfm` as a function of two arguments, `l` and `s`, each of type `TyAddConst(W, TyIntNonNeg)`. The function's return value in this case is computed by `ExprBinop` which applies a binary operator to two values (here the two function arguments `l`, `s`) and returns a value of type `TyAddConst(W, TyIntNonNeg)`. Semigroups are a special case of binary operators in which the two arguments and the return value are all of the same type, so we can put the `SgAddOmega` semigroup inside the `ExprBinop`. (ERL handles semigroups as a special case to better match the definition of RAML, in which semigroups are important due to supporting algebraic properties that would not apply to more general binary operators.)

The core of the example's transform is the `SgAddOmega` and `SgIntPlus`, which mean it will add its arguments as integers and treat the `W` constant as an infinity, matching the algebraic expression from Section 3.1.1. The rest of the ERL transform syntax is just setting up the types.

Expressions such as `ExprInt(10)` are simply a constant value. Expressions of the form *name* (in this example the "`l`" and "`s`" in the `ExprBinop`) refer to arguments of `fun`s, with standard rules of lexical scoping. `ExprApply` takes a `fun` and applies a list of expressions as arguments; in principle this is equivalent to the `fun`'s expression where any *name* argument references are replaced by the corresponding argument expression of the `ExprApply` – that is, `ExprApply(fun (x) -> `e_1`(x), `e_2`)` is equivalent to $e_1(e_2)$ – so it adds no expressive power to the language, but it simplifies the specification of programs based on nested transforms and allows common subexpressions to be factored out.

3.1.4 Compound types

The most complex types are `TyRecord` and `TyUnion`, which contain a set of any number of name/type pairs. For example, the ERL code in Figure 3.5 defines a record type `sig` with three fields. A value of this type is a tuple with three elements. An important subtlety of the language's design is that the order of field definitions is irrelevant; the type is equal to the one specified as `sigcanonical`.

```
let type sig =
  TyRecord(
    dist = TyAddConst(W, TyIntNonNeg),
    bw = TyIntNonNeg,
    path = TyListSimp(TyInt)
  )
```

```
let type sigcanonical =
  TyRecord(
    bw = TyIntNonNeg,
    dist = TyAddConst(W, TyIntNonNeg),
    path = TyListSimp(TyInt)
  )
```

Figure 3.5: ERL record type, and equivalent type with canonical field order.

We define the canonical form of a record or union type as the form in which the fields are sorted alphabetically by name. (Duplicate field names are forbidden, so there is always a single canonical form.) The type definitions, and most constructors over compound types – `PoRecord`, `PoUnion`, `SgRecord`, `SgUnion`, `ExprRecord`, `ExprSwitch` – are converted to canonical form after being parsed.

One exception is `PoRecordLex`, in which the field sequence is critical: it defines a lexicographic order over the fields, so the first field specified in the syntax takes precedence over the second and so on. For example, given the type

```
TyRecord(a=TyInt, b=TyIntPos),
```

the preorders

```
PoRecordLex(a = PoIntLte:TyInt, b = PoIntLte:TyIntPos)
PoRecordLex(b = PoIntLte:TyIntPos, a = PoIntLte:TyInt)
```

implement the left- and right-lexicographic orders respectively over the same type. If we did not have the canonicalisation of types then these orders would be over incompatible types, restricting the flexibility of the language.

Figure 3.6 illustrates the result of parsing the following example:

```
let preorder ord =
  PoRecordLex(
    dist = PoAddTop(PoIntLte : TyIntNonNeg) : TyAddConst(W, TyIntNonNeg),
    bw = PoDual(PoIntLte : TyIntNonNeg) : TyIntNonNeg,
    path = PoListLenLte : TyListSimp(TyInt)
  ) : sig
```

where `sig` is the same type defined earlier. (`path` is omitted from the diagram to save space.) There are effectively two parallel parse trees: one for the preorder and one for its type, with each preorder node corresponding to a subtree of the type parse tree. The child nodes of `TyRecord` are labelled with field names but are unordered, whereas

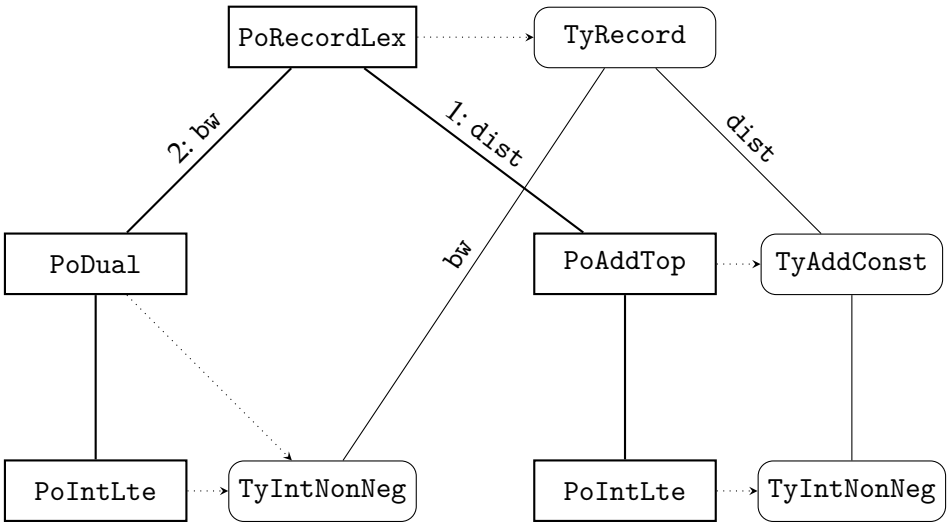


Figure 3.6: Linked parse trees of preorder and type.

the child nodes of `PoRecordLex` are annotated with their order in the lexicographic comparison. `PoDual` has the same type as the preorder it is applied to, so it shares the `PoIntLte`'s type node.

The other compound type is `TyUnion` which represents a disjoint union type, such as:

```
let type sig =
  TyUnion(
    external = TyInt,
    internal = TyIntRange(0, 15)
  )

let constant c1 = ExprInject(external, ExprInt(100):TyInt):sig

let constant c2 = ExprInject(internal, ExprInt(10):TyIntRange(0, 15)):sig
```

Values of this type are tagged as either `external` or `internal`; the `ExprInject` expression represents a value that has been injected into one of the tagged sets of a union type.

The algebraic semantics of a record type is simply the product of its field types:

$$[\![\texttt{TyRecord}(n_1{=}t_1,\ n_2{=}t_2,\ \ldots)]\!] = [\![t_1]\!] \times [\![t_2]\!] \times \ldots$$

where the `TyRecord` is assumed to be in canonical form. Values of this type are tuples, containing one value for each of the record's fields. The semantics of a union type is similarly the disjoint union of its field types.

Note that this throws away the field names entirely – they are not part of the routing algebra that the specification represents. However, they are important for the practical implementation of the routing algebra: users of the protocol need meaningful names to refer to the various fields. When we describe an approach to router configuration languages in Chapter 4 we use the ERL field names as part of the command-line user interface, but they have no effect on the fundamental behaviour of the routing language so we do not need them in the algebra.

The full syntax and algebraic semantics of ERL are defined in Appendix A. In addition to the features introduced in the previous section, it includes string, list, set and min-set types, along with a range of other preorders, semigroups and transforms.

3.2 RAML

Whereas ERL is a fairly low-level language with verbose syntax and explicit types, and consists of individual semigroups and preorders and transforms, RAML is based around higher-level algebraic structures. The semantic domain of this version of RAML is the order transform algebras:

$$(S,\ L,\ \preceq,\ \triangleright).$$

(A more general version of RAML supports algebras from the other quadrants discussed in Chapter 2, but they are not necessary for the focus of this dissertation. The definition of RAML in this chapter should be seen as a subset of the 'real' RAML; the full language design is ongoing work and not covered here, but this subset covers the important concepts and features that are necessary for an understanding of the full system.)

The semantics of a RAML expression also includes a set of algebraic properties: whether it is *increasing* or not; whether it is *distributive* or not; whether it has an *infinity* metric and, if so, the value of that infinity; whether it has an *identity* label and, if so, its value; and many other properties that are either desired by the routing algorithms, or are necessary for the property inference rules. Every pre-defined base expression in RAML is associated with a known set of properties, and every constructor expression has inference rules to compute its properties based on the properties of its sub-expressions.

The example from Section 3.1.1 can be expressed as the algebra

$$(\mathbb{N}^\infty,\ \mathbb{N}^\infty,\ \leq,\ +)$$

where "$\leq$" and "$+$" treat the value ∞ in the standard numeric way, corresponding to the ω or `W` in the ERL examples. This is the basic shortest paths algebra, *sp*, for which various algebraic properties are easily discovered: it is increasing and distributive, with infinity metric ∞ and identity label 0.

```
raml-base ::= int_min_plus(erl-ty-int-unbounded)
            | int_range_min_plus(num, num)
            | int_max_min(erl-ty-int)
            | paths(erl-ty)
            | strata(num, num, num...)

raml-exp  ::= raml-base
            | add_top(name, raml-exp)
            | lex_product(name=raml-exp, name=raml-exp, ...)
            | function_union(name=raml-exp, name=raml-exp, ...)
            | right(raml-exp)
            | left(raml-exp)
```

Figure 3.7: RAML grammar.

The syntax of RAML is defined by the grammar in Figure 3.7. Note that we reuse ERL's syntax for types (*erl-ty* etc.) – we could provide a more self-contained definition of RAML by duplicating the relevant type specifications, but we choose to avoid that redundancy here. We distinguish RAML and ERL syntax by writing keywords with `CamelCaseCapitalisation` for ERL and `underscore_separated_words` for RAML, to reduce confusion when mixing the two.

We will express the algebraic semantics of a RAML expression as

$$\begin{aligned}
[\![\texttt{int_min_plus(TyInt)}]\!] &= ((\mathbb{Z}, \mathbb{Z}, \leq, +), \\
&\quad \{\text{INCR} = \bot, \text{DIST} = \top, \text{TOP} = \bot, \text{BOT} = \bot\}) \\
[\![\texttt{int_min_plus(TyIntNonNeg)}]\!] &= ((\mathbb{N}, \mathbb{N}, \leq, +), \\
&\quad \{\text{INCR} = \top, \text{DIST} = \top, \text{TOP} = \bot, \text{BOT} = 0\}) \\
[\![\texttt{int_min_plus(TyIntPos)}]\!] &= ((\mathbb{N}^+, \mathbb{N}^+, \leq, +), \\
&\quad \{\text{INCR} = \top, \text{DIST} = \top, \text{TOP} = \bot, \text{BOT} = 1\})
\end{aligned}$$

including the algebra and the associated set of properties. In this case we must handle each possible argument type explicitly as the properties differ for each type. The properties INCR (increasing) and DIST (distributive) can be in one of three states: true ($\top$), false ($\bot$), or unknown (omitted from the properties set entirely; ideally the language will be designed to ensure no properties will ever be unknown, but at the current stage of development this is not guaranteed). The TOP property is an element that is a top of the preorder (or $\bot$ if there is known to be no such element, or omitted if unknown). In general terms this element is a *witness* to a separate property such as HAS_TOP (i.e. a value which demonstrates the truth of that property), and many other properties will also have witnesses (e.g. a pair (l, s) that demonstrates a violation of INCR). We will ignore most witnesses since they are not relevant when compiling to ERL, but we make the values TOP and BOT explicit as some routing algorithms will require a compiled

ERL representation of them. Many other properties are important for the completeness of RAML, but are omitted here for simplicity.

`paths` implements the path algebra described in Section 2.2.6 over some arbitrary identifier type, with the semantics

$$\begin{aligned}[\![\texttt{paths}(t)]\!] \;=\; & ((([\![t]\!]^*_{simp} \cup \{\infty\},\ [\![t]\!] \times [\![t]\!],\ \preceq_{listLen},\ \rhd_{paths}),\\ & \{\text{INCR} = \top,\ \text{DIST} = \bot,\ \text{TOP} = \infty,\ \text{BOT} = []\})\end{aligned}$$

where $\preceq_{listLen}$ and $\rhd_{paths}$ are as defined before.

`strata` is a generalisation of the customer–provider–peer algebra also described in Section 2.2.6, named for its association with the Stratified Shortest-Paths Problem [Gri10]. We take the first argument as the number of metric values, the second as the number of label values, and the remaining arguments as the contents of the matrix defining $\rhd$. For example, *cpp* was defined with the matrix

$\rhd_{cpp}$	C	R	P	∞
c	C	∞	∞	∞
r	R	∞	∞	∞
p	P	P	P	∞

and would be expressed in RAML as

```
strata(4, 3,
  0, 3, 3, 3,
  1, 3, 3, 3,
  2, 2, 2, 3)
```

with the metrics and labels being mapped onto integers. (This can be made more user-friendly in practice by supporting enumeration types in ERL and using meaningful identifiers instead of numbers.) The algebra's $\preceq$ is simply integer $\leq$. The algebraic properties can be determined automatically from any matrix by enumerating all combinations to see what properties hold. The matrix for this *cpp* example can easily be extended to support the full range of policy functions for strata explored previously by Griffin [Gri10].

The semantics of a constructor expression *raml-exp* are defined in terms of the semantics of the sub-expressions. For example, the two-argument `lex_product` has the semantics defined in Figure 3.8. `lex_products` over more than two arguments are a natural extension of this.

The INCR and DIST properties depend in potentially complex ways on the properties of $[\![e_1]\!]$ and $[\![e_2]\!]$, as discussed elsewhere [GG07], but importantly they depend *only* on those properties – there is no need for any deeper analysis of the sub-expressions to

$$
\begin{aligned}
[\![\texttt{lex\_product}(n_1{=}e_1,\ n_2{=}e_2)]\!] \;=\; & ((S_1 \times S_2,\ L_1 \times L_2,\ \preceq_1 \vec{\times} \preceq_2,\ \rhd_1 \times \rhd_2), \\
& \{\text{INCR} = \ldots,\ \text{DIST} = \ldots, \\
& \text{TOP} = \begin{cases} \bot & \text{if } t_1 = \bot \vee t_2 = \bot \\ (t_1, t_2) & \text{otherwise} \end{cases}, \\
& \text{BOT} = \begin{cases} \bot & \text{if } b_1 = \bot \vee b_2 = \bot \\ (b_1, b_2) & \text{otherwise} \end{cases} \})
\end{aligned}
$$

where

$$
\begin{aligned}
[\![e_1]\!] &= ((S_1, L_1, \preceq_1, \rhd_1), \{\text{TOP} = t_1,\ \text{BOT} = b_1,\ \ldots\}) \\
[\![e_2]\!] &= ((S_2, L_2, \preceq_2, \rhd_2), \{\text{TOP} = t_2,\ \text{BOT} = b_2,\ \ldots\})
\end{aligned}
$$

Figure 3.8: Semantics of the `lex_product` RAML expression.

$$
\begin{aligned}
[\![\texttt{add\_top}(c,\ e)]\!] &= ((\{c\} \cup S',\ \{c\} \cup L',\ \preceq,\ \rhd), \\
&\quad \{\text{INCR} = (i' \wedge t' = \bot),\ \text{DIST} = d',\ \text{TOP} = c,\ \text{BOT} = b'\}) \\
\text{where} & \\
[\![e]\!] &= ((S',\ L',\ \preceq',\ \rhd'), \\
&\quad \{\text{INCR} = i',\ \text{DIST} = d',\ \text{TOP} = t',\ \text{BOT} = b'\}) \\
\text{and} & \\
c \rhd s &= c \\
l \rhd c &= c \\
l \rhd s &= l \rhd' s \quad \text{for all } s \in S',\ l \in L' \\
\text{and} & \\
c \preceq s & \\
s_1 \preceq s_2 &\Leftrightarrow s_1 \preceq' s_2 \quad \text{for all } s, s_1, s_2 \in S'
\end{aligned}
$$

Figure 3.9: Semantics of the `add_top` RAML expression.

determine the properties of the `lex_product`, allowing a straightforward constructive approach to compiling and verifying RAML programs.

In RAML's `lex_product` we have the same field ordering issue as in ERL: the fields must be shuffled into a canonical order for the types, but we should change $\vec{\times}$ into $\overleftarrow{\times}$ so that it compares fields in the sequence specified by the syntax.

We can similarly define the semantics of `add_top`, which adds new metric and label values that act as the top element in the order and as an annihilator element in the transform, as in Figure 3.9.

In parallel with the algebraic semantics, each RAML expression corresponds to a collection of ERL expressions. This fully defines the behaviour of the RAML-to-ERL com-

```
sig = TyAddConst(c, (|e'|)_sig)
lbl = TyAddConst(c, (|e'|)_lbl)
ord = PoAddTop((|e'|)_ord)
tfm = fun (l s) ->
        ExprConstCase(l, c, c, fun (l2) ->
          ExprConstCase(s, c, c, fun (s2) ->
            ExprApply((|e'|)_tfm, l2, s2)))
```

Figure 3.10: ERL translation of RAML `add_top` constructor.

piler. We express the ERL translation of a RAML expression e as $(\!|e|\!)$. As we are using RAML to represent order transforms, the ERL collections always have a signature type `sig`, label type `lbl`, order `ord` over signatures, and transform `tfm`. We use the notation "$(\!|e|\!)_{\texttt{sig}}$" to refer to the `sig` component of the ERL semantics $(\!|e|\!)$.

In principle, the algebraic semantics defined for RAML will be isomorphic to the semantics defined for the ERL that is derived from the RAML. This is critical for the correctness of the metarouting system: we cannot determine algebraic properties of ERL expressions directly, but if we can prove algebraic properties based on RAML expressions, and show that we extract the same routing algebra after compiling the RAML to ERL, then the same algebraic properties will apply to the ERL code. Without this guarantee, bugs in the definition of the RAML-to-ERL compiler could easily violate the correctness conditions that the algebraic properties are meant to ensure.

The desired equivalence between algebraic semantics of RAML and ERL is then

$$\begin{aligned}
[\![e]\!] &= ((S, L, \preceq, \triangleright), \{\ldots\}) \\
[\![(\!|e|\!)_{\texttt{sig}}]\!] &= S \\
[\![(\!|e|\!)_{\texttt{lbl}}]\!] &= L \\
[\![(\!|e|\!)_{\texttt{ord}}]\!] &= \preceq \\
[\![(\!|e|\!)_{\texttt{tfm}}]\!] &= \triangleright
\end{aligned}$$

where $[\![\ldots]\!]$ is overloaded to mean both "algebraic semantics of ERL expression" and "algebraic semantics of RAML expression".

For example, $(\!|\texttt{add\_top}(c,\ e')|\!)$ is the collection of ERL expressions in Figure 3.10. Applying the ERL algebraic semantics defined in Appendix A to these expressions, we get the algebras in Figure 3.11. This can be compared against the algebraic semantics we previously defined directly for the RAML expression `add_top`: the order and transform are written in slightly different but equivalent ways, confirming that the defined semantics for RAML-to-ERL-to-algebra and RAML-to-algebra are compatible. A more formal and comprehensive approach (not attempted here) should prove this equivalence for every case, allowing the correctness of the whole RAML-to-ERL compiler to be verified.

$$
\begin{aligned}
[\![\texttt{sig}]\!] &= \{c\} \cup [\![(\!|e'|\!)_{\texttt{sig}}]\!] \\
[\![\texttt{lbl}]\!] &= \{c\} \cup [\![(\!|e'|\!)_{\texttt{lbl}}]\!] \\
[\![\texttt{ord}]\!] &= \preceq \text{ where } s_1 \preceq s_2 \Leftrightarrow \\
&\qquad s_2 = c \ \vee \ (s_1 \neq c \wedge s_2 \neq c \wedge s_1 \, [\![(\!|e'|\!)_{\texttt{ord}}]\!] \, s_2) \\
[\![\texttt{tfm}]\!] &= \lambda ls. \begin{cases} c & \text{if } l = c \\ c & \text{if } s = c \\ [\![(\!|e'|\!)_{\texttt{tfm}}]\!](l,s) & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.11: Algebraic semantics of ERL translation of RAML `add_top` constructor.

Appendix B gives a definition of the algebraic and ERL semantics of all the RAML expressions introduced in our RAML syntax.

3.3 Language examples

3.3.1 Distance-bandwidth

We start with a simple routing language expressed in RAML:

```
lex_product(
  dist = add_top(W, int_min_plus(TyIntNonNeg)),
  bw = int_max_min(TyIntNonNeg)
)
```

This is designed to implement the algebra

$$(\mathbb{N},\ \mathbb{N},\ \geq,\ \min) \overleftarrow{\times} (\mathbb{N}^\infty,\ \mathbb{N}^\infty,\ \leq,\ +).$$

(Note that the algebra is based on the canonical ordering of the RAML record, where `bw` comes before `dist`, with the right-lexicographic product $\overleftarrow{\times}$ instead of the more common $\vec{\times}$ so that `dist` takes precedence in the order.)

Based on Appendix B we can extract the algebraic semantics of the RAML expression, which will give an equivalent algebra but with ∞ renamed to `W` and handled explicitly by the orders and transforms.

Separately, we can extract an ERL specification from the RAML using the same table, giving Figure 3.12. We can then use Appendix A to determine the algebraic semantics of this ERL specification, presented in Figure 3.13 with some algebraic simplifications.

```
lbl = TyRecord(dist=TyAddConst(W, TyIntNonNeg), bw=TyIntNonNeg)
sig = TyRecord(dist=TyAddConst(W, TyIntNonNeg), bw=TyIntNonNeg)
ord = PoRecordLex(dist=PoAddTop(PoIntLte), bw=PoDual(PoIntLte))
tfm = fun (l s) ->
    ExprRecord(
      dist = ExprApply(
        fun (l1 s1) -> ExprBinop(BoSemigroup(SgAddOmega(SgIntPlus)), l1, s1),
        ExprSelect(l, dist), ExprSelect(s, dist)),
      bw = ExprApply(
        fun (l2 s2) -> ExprBinop(BoSemigroup(SgIntMin), l2, s2),
        ExprSelect(l, bw), ExprSelect(s, bw))
    )
```

Figure 3.12: ERL code for distance-bandwidth example. (Type annotations are omitted for clarity.)

$$
\begin{aligned}
\texttt{sig} &= \mathbb{N} \times (\{\texttt{W}\} \cup \mathbb{N}) \\
\texttt{lbl} &= \mathbb{N} \times (\{\texttt{W}\} \cup \mathbb{N}) \\
\texttt{ord} &= (u_b, u_d) \preceq (v_b, v_d) \Leftrightarrow \\
&\qquad ((v_d = \texttt{W} \vee u_d \leq v_d) \wedge \neg(u_d = \texttt{W} \vee v_d \leq u_d)) \ \vee \\
&\qquad ((v_d = \texttt{W} \vee u_d \leq v_d) \wedge (v_b \leq u_b)) \\
\texttt{tfm} &= \lambda(l_b, l_d)(s_b, s_d).\ \left(l_b \min s_b, \left\{ \begin{array}{ll} \texttt{W} & \text{if } s_d = \texttt{W} \\ \texttt{W} & \text{if } l_d = \texttt{W} \\ l_d + s_d & \text{otherwise} \end{array} \right\} \right)
\end{aligned}
$$

Figure 3.13: Algeraic semantics of ERL code for distance-bandwidth example.

The design of RAML and ERL should ensure the routing algebra (`sig`, `lbl`, `ord`, `tfm`) is isomorphic to the $(S,\ L,\ \preceq,\ \triangleright)$ algebra that was extracted directly from the RAML.

3.3.2 Compiling to C++

The ERL specification defines an executable program; an interpreter that implements all the types and expressions could easily perform the routing language's computations. For optimal performance, we instead compile ERL into C++ code to perform the same computations.

The details of the compiler are discussed by Billings [Bil09]. Here we will briefly demonstrate its output for this example, to help explain the full behaviour of the com-

pilation process.

The following listing gives the generated C++ code, with some comments inserted manually. The code is fairly opaque and not intended to be especially human-readable, but some details can be highlighted. One important point to note is that no functions are generated for e.g. printing metric values or marshalling (encoding to a sequence of bytes): the output is purely declarative, using C++'s template system to define a type `sig` that encodes the metric structure. It therefore has a close structural correspondence to the ERL code we extract from the RAML specification, though with more convoluted syntax.

One of the more complex correspondences between ERL and C++ is record types. C++ templates cannot easily handle variable numbers of parameters, so records are implemented with a `RecCons` cell per field: each is of the form `RecCons<`*name*`,` *type*`,` *rest*`>` where *rest* is either another `RecCons` or (for the final field) `RecEnd`. The outermost `RecCons` is then wrapped in a `RecWrap` type due to some implementation details.

The C++ template system is effectively a Turing-complete functional programming language (albeit with severe implementation restrictions) that is executed at compile-time to hook together pieces of run-time-executable C++ code. In our case, the C++ compiler handles the job of combining the type declarations with a library of generic printing/marshalling/etc. code to produce a very efficient implementation of all the necessary functionality. This library, named *libmrc*, defines the C++ types `AddConst`, `IntBigNonNeg` and so on.

As a very brief summary of important C++ syntax: "`typedef Foo ty0`" binds the existing type `Foo` to the new type name `ty0`. The type "`Foo<S, T>`" is the generic templated type `Foo`, instantiated with parameters that are the types `S` and `T`. For transform expressions, "`struct f { t operator()(...) { ... } }`" defines a kind of function (specifically a functor type) named `f` that returns a value of type `t`.

```
// String constants
std::string str_W("W");
std::string str_bw("bw");
std::string str_dist("dist");

// Define the 'sig' record type
typedef AddConst<IntBigNonNeg, str_W> ty0;
typedef RecCons<str_dist, ty0, RecLast> ty1;
typedef RecCons<str_bw, IntBigNonNeg, ty1> ty2;
typedef RecWrap<ty2> sig;

// Define 'lbl' which happens to share the type 'sig'
typedef sig lbl;
```

```
// Define nested transform functions for ‘tfm’
struct funct
{
  ty0 operator()(ty0 l1, ty0 s1) const
  {
    // Apply the addition function to l1, s1
    typedef AddConstOmega<IntBigNonNegPlus> ty4;
    ty0 var7(ty4()(l1, s1));
    return var7;
  }
};

struct funct1
{
  IntBigNonNeg operator()(IntBigNonNeg l2, IntBigNonNeg s2) const
  {
    // Apply the min function to l2, s2
    IntBigNonNeg var17(IntBigNonNegMin()(l2, s2));
    return var17;
  }
};

struct tfm
{
  sig operator()(lbl l, sig s) const
  {
    RecEnd var1;

    // Add the ‘dist’ fields
    typedef RecOpGet<1, ty0> ty5;
    ty0 var3(ty5()(s));
    ty0 var5(ty5()(l));
    funct var10;
    ty0 var2 = var10(var5, var3);

    // Cons the result with RecEnd
    typedef std::pair<ty0, RecEnd> ty6;
    ty6 var11 = std::make_pair(var2, var1);

    // Min the ‘bw’ fields
    typedef RecOpGet<0, IntBigNonNeg> ty7;
    IntBigNonNeg var13(ty7()(s));
```

```
    IntBigNonNeg var15(ty7()(l));
    funct1 var20;
    IntBigNonNeg var12 = var20(var15, var13);

    // Cons the result with the previous RecCons
    typedef std::pair<IntBigNonNeg, ty6> ty8;
    ty8 var21 = std::make_pair(var12, var11);

    // Construct and return the new metric record
    sig var0(var21);
    return var0;
  }
};

// Define the 'ord' lexicographic-product preorder
typedef AddConstTop<IntBigLte> ty9;
typedef PolyDual<IntBigLte> ty10;
typedef RecOrdLexCons<0, IntBigNonNeg, ty10, RecOrdLast> ty11;
typedef RecOrdLexCons<1, ty0, ty9, ty11> ty12;
typedef RecOrdWrap<ty12> ord;
```

Despite the heavy use of C++ template types and functors, this can compile into efficient machine code due to function inlining optimisations: if `IntBigNonNeg` is replaced with bounded integers (as would be common in a routing language designed for practical usage) to avoid an algebraically correct but inevitably slower reliance on an external arbitrary-size-integer package, then the GCC compiler with the `-O2` optimisation flag implements `tfm::operator()(lbl l, sig s)` in about 80 assembly instructions on x86 (plus a bit more for internal error checking).

Given all these definitions, we can write a simple standalone C++ program to manipulate metric and policy values, such as:

```
#include "dist_bw-generated-output.hpp"
int main()
{
  sig m0 = sig(parse_sig("<dist=1, bw=1000>"));
  std::cout << "Input metric: " << m0 << "\n";

  lbl l0 = lbl(parse_sig("<dist=10, bw=500>"));
  sig m1 = tfm()(l0, m0);
  std::cout << "Output metric: " << m1 << "\n";
    // prints the string "<dist=11, bw=500>"
}
```

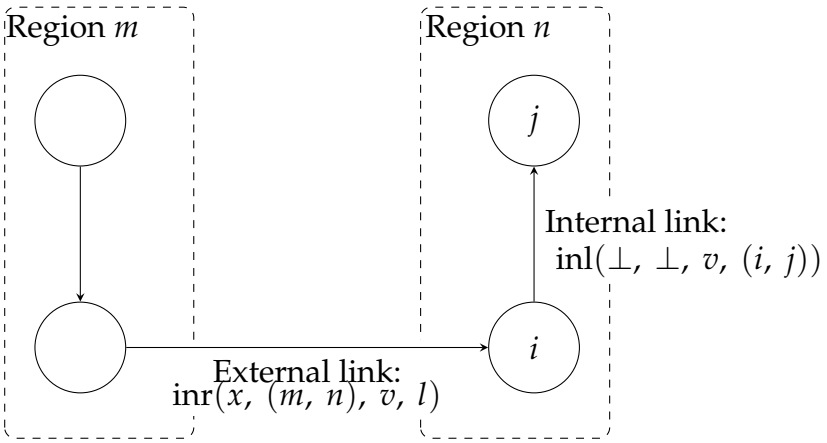


Figure 3.14: A network with link weights from the scoped product algebra.

Apart from the input strings describing the initial metric and label, this code is completely independent of the routing language that we specified: it simply relies on the abstracted interface of `sig`, `lbl` and `tfm` generated by mrc (the ERL-to-C++ compiler). This abstraction is the basis of how the metarouting toolkit separates the language implementation and algorithm implementation. Instead of manually writing a C++ program like this, the generated output is compiled against a standard gQuagga wrapper program that exposes the routing language's behaviour to the algorithm implementation.

3.3.3 Scoped product

As a more advanced example, we will demonstrate the metarouting system with an algebra for the network in Figure 3.14. The goal is to implement a two-level partitioning scheme within the routing language. (This contrasts to most current routing protocols which make the levels an integral part of the algorithm, as with IBGP vs. EBGP – we will gain flexibility (for example we could easily extend this to more than two levels) but may lose the performance optimisations that are possible when the algorithm knows more about the network structure.)

We use the RAML specification of Figure 3.15. `cpp` is a reference to the customer–provider–peer instance of `strata`, abbreviated here for readability. The aim is that internal links will be labelled with an internal distance and internal edge identifier (to detect loops as with BGP's AS paths, but inside the region instead of only outside). The external customer–provider–peer commercial relationship and the external path will not be modified at all while passing over internal links, so `right` is used. External links will be labelled with customer–provider–peer and external edge identifier. External links will also reset the two internal attributes to some default value, using `left`, so

```
add_top(W,
  function_union(
    internal =
      lex_product(
        ecomm = right(cpp),
        epath = right(paths(TyIntNonNeg)),
        idist = int_min_plus(TyIntNonNeg),
        ipath = paths(TyIntNonNeg)
      ),
    external =
      lex_product(
        ecomm = cpp,
        epath = paths(TyIntNonNeg),
        idist = left(int_min_plus(TyIntNonNeg)),
        ipath = left(paths(TyIntNonNeg))
      )
  )
```

Figure 3.15: Scoped product routing language specification.

that internal details will not leak out of a region.

Extracting the routing algebra $(S,\ L,\ \preceq,\ \triangleright)$ for this input is simply a matter of recursively applying the semantic rules for $[\![raml\text{-}exp]\!]$.

We find the metric type S is

$$(\{0,\ 1,\ 2,\ 3\} \times (\mathbb{N}^*_{simp} \cup \{\omega\}) \times \mathbb{N} \times (\mathbb{N}^*_{simp} \cup \{\omega\})) \cup \{\omega\}.$$

The order $\preceq$ is the lexicographic order on the metric components. The label type is a disjoint union, being either a value

$$\text{inl}(\mathbf{1} \times \mathbf{1} \times \mathbb{N} \times (\mathbb{N} \times \mathbb{N}))$$

for internal links, or

$$\text{inr}(\{0,\ 1,\ 2\} \times (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \times (\mathbb{N}^*_{simp} \cup \{\omega\}))$$

for external links, or ω for unusable links.

$\triangleright$ is determined as

$$\begin{aligned}
\omega \triangleright s &= \omega \\
l \triangleright \omega &= \omega \\
\text{inl}(\bot,\ \bot,\ v,\ (i,\ j)) \triangleright (ec,\ ep,\ id,\ ip) &= (ec,\ ep,\ v + id,\ (i,\ j) \triangleright_{paths} ip), \\
\text{inr}(x,\ (m,\ n),\ v,\ l) \triangleright (ec,\ ep,\ id,\ ip) &= (x \triangleright_{cpp} ec,\ (m,\ n) \triangleright_{paths} ep,\ v,\ l).
\end{aligned}$$

(One minor bug in this definition is that loops in path attributes will result in metrics such as $(0,\ \omega,\ 0,\ [])$, not quite the desired ω. We have not yet implemented a complete solution for this, so we neglect it from this example. Section 8.1.2 discusses some possible solutions in more detail.)

Compiling the RAML code ERL gives the metric type

```
sig =
  TyAddConst(W,
    TyRecord(
      ecomm = TyIntRange(0, 3),
      epath = TyAddConst(NOTSIMPLE, TyListSimp(TyIntNonNeg)),
      idist = TyIntNonNeg,
      ipath = TyAddConst(NOTSIMPLE, TyListSimp(TyIntNonNeg))
    )
  )
```

i.e. either the constant `W` or a record with four fields, where the path fields are simple (duplicate-free) lists of integers or the constant `NOTSIMPLE`, and the other fields are just integers. The order over the metric type is

```
ord =
  PoAddTop(
    PoRecordLex(
      ecomm = PoIntLte,
      epath = PoAddTop(PoListLenLte),
      idist = PoIntLte,
      ipath = PoAddTop(PoListLenLte)
    )
  )
```

so `W` is less preferred than any other value, and other values are compared with a lexicographic product over the fields, with path fields being compared by length to prefer shorter paths. The label type is

```
lbl =
  TyAddConst(W,
    TyUnion(
      internal = TyRecord(
        ecomm = TyUnit,
        epath = TyUnit,
        idist = TyIntNonNeg,
        ipath = TyIntNonNeg
      ),
```

```
      external = TyRecord(
        ecomm = TyIntRange(0, 2),
        epath = TyIntNonNeg,
        idist = TyIntNonNeg,
        ipath = TyAddConst(NOTSIMPLE, TyListSimp(TyIntNonNeg))
      )
    )
  )
```

using a union to distinguish between internal labels and external labels. Internal labels do not specify a value for the external fields, so `TyUnit` is used; we could optimise the ERL code by removing these fields entirely as they carry no data. Internal labels specify a single integer for `ipath` which will be prepended to the metric's path, whereas external labels specify a whole list which will replace that path (due to the use of `left` in the RAML).

The policy application function is the most complex expression:

```
tfm = fun (l s) ->
  ExprConstCase(l, W, W, fun (l2) ->
    ExprConstCase(s, W, W, fun (s2) ->
      ExprApply(fun (l3 s3) ->
        ExprSwitch(l3,
          internal = fun (l4) -> ExprRecord(
            ecomm = ExprSelect(s3, ecomm),
            epath = ExprSelect(s3, epath),
            idist = ExprBinop(BoSemigroup(SgIntPlus),
                      ExprSelect(l4, idist), ExprSelect(s3, idist)),
            ipath = ExprBinop(BoListCons,
                      ExprSelect(l4, ipath), ExprSelect(s3, ipath))
          ),
          external = fun (l4) -> ExprRecord(
            ecomm = ...,
            epath = ExprBinop(BoListCons,
                      ExprSelect(l4, epath), ExprSelect(s3, epath)),
            idist = ExprSelect(l4, idist),
            ipath = ExprSelect(l4, ipath)
          )
        ), l2, s2
      )
    )
  )
```

First, the `ExprConstCase` expressions return `W` if `l` or `s` are already `W`. Otherwise, their values (now known not to be `W`) are bound to the names `l3` and `s3` and the main body of

the transform is executed. The `ExprSwitch` checks whether the label is tagged as internal or external, then binds the label value (minus the tag) to `l4`. Finally the transform returns the appropriate `ExprRecord` value, whose fields are calculated by selecting the fields of the label `l4` and the metric `s3` and then either copying them into the record or applying a binary operator to them.

(The implementation of $\triangleright_{cpp}$ in the `external ecomm` field is omitted as we do not yet have a concise way to define it in ERL.)

Chapter 4

Generalising vector protocols

In Chapter 1 we described how the algebraic model of routing separates the *language*, that determines how a network configuration can be expressed and the meaning of 'best' paths over that network, from the *algorithm* that computes the best paths of a given network configuration. We have also described how metarouting builds on this concept by providing a method for *specification* of routing languages, that is designed to be used for both modeling and implementing practical internet routing protocols.

There is an important question here: How well does this language/algorithm split apply to current routing protocol designs and implementations? To what extent is it an *observation* of today's routing from a new perspective (providing a solid foundation from which we can explore new ideas), rather than a radical new paradigm that requires a clean-slate approach to routing? If it is a natural division then we gain two benefits. Firstly, we can use this model to help understand current protocol designs. It is much easier to reason about the protocol's language semantics when it is separated from the low-level algorithm implementation details, but only if this is a correct and clean separation and does not introduce significant new complexities itself. Secondly, we can use current protocol implementations as a basis for a metarouting implementation. If we can cleanly separate the language and algorithmic components of the implementation, then we can rewrite the language component (where the complexity is largely in the conceptual design, which can be attacked with the tools provided by the metarouting approach) without having to change the algorithm (where the complexity is largely in the tens of thousands of lines of code dealing with network packets and maintaining the routing tables and forwarding tables and providing a configuration interface, which is much less amenable to a theory-based attack).

Some research has explored ways to improve the task of implementing routing algorithms. XORP [HKG+05] is based around composable modules with event-driven interfaces, to provide a clean architecture and extensibility that supports experimentation with new routing protocols. Each module maintains internal state and processes a flow

of routes. For example, BGP can be decomposed into separate modules that communicate with a peer, apply filters, then resolve next-hop information, repeated in parallel for each peer (dynamically adding and removing sets of modules as peers connect and disconnect), all feeding into a single decision process module that picks best routes, before fanning out into parallel sets of modules to filter and distribute best routes back to peers. Viewed with our language/algorithm split in mind, XORP is modularising the algorithm implementation while the language implementation remains a complex chunk of C++ within the BGP decision process module; it is orthogonal and complementary to metarouting's approach of specifying the language at a higher level and compiling into the implementation.

Declarative routing [LHSR05] implements routing protocols in a declarative database query language, Datalog, instead of the traditional imperative C/C++ languages. Protocols are specified as queries in the language, crafted so that the execution of the query in a distributed Datalog implementation matches the desired behaviour of the protocol. Examples given in the paper have a clear language/algorithm split, with the language providing a pair of functions (*AGG* and *f_compute*). As with XORP, it aims to provide extensibility and flexibility to algorithm implementations; whereas XORP does this by providing a system architecture to manage the complexity of the implementation, declarative routing works by pushing all of the complexity down into the Datalog layer. As with metarouting, routing languages are specified in a declarative language; however, whereas RAML has well-defined algebraic semantics and automatic property inference to ensure the routing language can be safely used, Network Datalog has far more complex semantics and the safety properties may not be provable in general. (Nigam et al. [NJW+10] list difficulties with specifying its semantics, and prove correctness only for the non-recursive fragment of the language.) As such, declarative routing would be more appropriate as an implementation language for routing algorithms in the metarouting model, as an easier-to-write alternative to C/C++, and does not reduce the need for RAML and routing algebras when defining routing languages.

With these approaches attempting to address the complexity of algorithm implementation, we will focus on metarouting's attempt to address the complexity of the routing languages.

In this chapter we start by formulating RIP and BGP in our language/algorithm terms, showing that they follow this model quite well but with a number of possible ways to perform the split. We then briefly discuss a number of other protocols, to determine whether the model applies more widely to protocol designs. Next we consider a few general issues with applying the algebraic routing model to vector protocols, because of the distributed nature of their computation and configuration.

Then we describe the details of implementing a metarouting system that reuses the algorithm implementations of RIP and BGP from the Quagga protocol suite, along

with the benefits and drawbacks of this approach, and consider the portability of this solution between different algorithm implementations.

4.1 Generalising RIP

4.1.1 Language

At a basic level, RIP is modelled algebraically as using the routing language of shortest paths over bounded integers: it is $(S,\ L,\ \preceq,\ \triangleright)$ where

$$\begin{aligned} S &= \{0, 1, 2, \ldots, 15, \infty\} \\ L &= \{1, 2, \ldots, 15, \infty\} \\ \preceq &= \leq \\ l \triangleright s &= \begin{cases} l + s & \text{if } l + s \leq 15 \\ \infty & \text{otherwise} \end{cases} \end{aligned}$$

This satisfies the *distributive* and *increasing* properties discussed earlier. Given this language, and a network configuration expressed as a labelled graph, we can use a range of algorithms – such as matrix multiplication, or a distributed Bellman-Ford implementation – and the properties guarantee they will converge to the same routing solution, without needing to reproduce the exact details of RIP's operation.

However, when looking deeper at the reality of RIP as used today, it becomes clear that this language is not the full story. In Quagga's RIP implementation, the main divergence from this basic model is policy expressed as *route maps*. The default processing on an interface (adding a positive integer to a route's metric before advertising that route) can be replaced entirely with a custom route map. These are functions that match on the metric value (equality with a configured constant), and on other properties of the route (interface, next-hop, IP address, route tag). They can filter out a matching route, change its next-hop or tag, and modify its metric by either adding a constant, subtracting a constant, or replacing the metric entirely.

The ability to apply different offsets depending on the exact value of the metric means we lose the guarantee of distributivity: the preference order between two routes can be flipped by the application of the same policy. This is not a major problem for RIP, since the distance vector algorithm will still converge to a local optimum solution.

More serious is the violation of the increasing property, by the ability to subtract or replace the metric. This is clearly a feature that must be used with great care and with knowledge of the network topology, to avoid persistent routing loops. Since one of the goals of the metarouting approach is to guarantee the absence of problems such

as routing loops without global constraints on configuration and topology, we simply cannot support this feature.

These issues are not unique to Quagga – they also apply to the implementations of RIP documented by Cisco IOS [Cis11] and JunOS [Jun10a]. We will discuss route maps in detail in Chapter 6, and ignore them for the rest of this chapter.

With the exception of route maps, the algebraic language expressed earlier is a quite accurate match to the operation of RIP implementations.

4.1.2 Algorithm

With a few complications, we can describe the routing language of RIP independently of its algorithm, as shown in the previous section. This section looks at the complementary question of how we can use RIP's algorithm independently of its routing language.

The RIP specification combines the algorithm and language into a single protocol definition, so it cannot be used unmodified with a new routing language. There are a number of ways it can be modified so that the algorithm definition is independent of (but imposes constraints on) the language. We have developed a specific set of modifications under the name *generalised RIP* (gRIP), which supports a wide range of routing languages without losing the essential character of RIP.

The first technical restriction in RIP is that metrics are represented in response packets as integers in the range $[1, 16]$ encoded in four bytes. Constraining routing languages to use exactly this metric would be extremely inflexible. Constraining them to use any metric that can be encoded in four bytes would allow more interesting variety (e.g. a combined distance–bandwidth metric with each number encoded in two bytes), but would still exclude many useful languages (e.g. one that stores a path within the metric, converting RIP from a distance-vector to a path-vector protocol). Relaxing the metric size to a per-language constant would allow further variety, but would require bounds on all lists and sets and would be inefficient if most of the allocated bytes are often unused.

gRIP aims for the most general approach of allowing variable-sized metrics (with the length encoded in the packet just before the metric), requiring of the routing language only that it provide functions for converting metric values to and from a stream of bytes. In practice, gRIP is not quite this general – it follows RIP's maximum packet size of 512 bytes with no ability to split a single route update across multiple packets, imposing an upper bound on the size of a single metric. Exceeding this bound results in a run-time error and the route is dropped from the advertisement packet.

A slightly more subtle restriction is in the assumptions the RIP algorithm design makes about the behaviour of its language. In particular, it assumes that counting-to-infinity

is an acceptable (though undesirable) state – this can occur whenever there is a cycle of three or more routers and one of them withdraws a route, depending on update timings. In standard RIP, the worst case is that it will take 15 update cycles (each up to 30 seconds long[1]) before the metrics associated with echoes of the original route reach 'infinity' and the route is dropped. The choice of 15 as the maximum metric is an explicit tradeoff between the speed of convergence in counting-to-infinity cases, and the maximum size of the network.

A routing language designed for use with gRIP must take this tradeoff into account. A shortest-path language $(\mathbb{N}^\infty, \mathbb{N}^\infty, \leq, +)$ over unbounded integers would be unsuitable as it would not guarantee convergence, even though it has a well-defined solution and can be computed correctly by other algorithms. But this does not restrict languages to the simple one used by RIP: it would be possible to combine a maximum hop count with an arbitrary metric in a lexicographic product, or to prevent cycles by encoding the path within the metric (as with BGP AS paths).

RIP's split horizon feature is designed to prevent counting-to-infinity cycles between two neighbours, and is purely an algorithmic optimisation with no additional impact on the routing language design, so we can use it in gRIP with no difficulties.

The poisoned reverse feature helps the propagation of information about route removals, by advertising the route with a metric of 16 to indicate removal instead of waiting for the expiry timeout. In gRIP we require the routing language to provide an infinity value, which is used for this purpose. We could instead have added a new flag to indicate withdrawal messages, but the use of an explicit infinity metric is more consistent with the original design of RIP.

4.2 Generalising BGP

BGP has a much more complex routing language than RIP, discussed in Section 2.3.2, and a detailed description is not attempted here. Fortunately, unlike RIP, the design of BGP naturally supports protocol extensions: route advertisements contain a series of variable-sized *attributes*, making up the complex metric type associated with each route. New attributes can be added by external agreement on the meaning of attribute type ID numbers, with no changes to the basic packet structure. This external agreement typically occurs through the publishing of RFCs and the allocation of numbers with IANA [ian11].

Most of the standard BGP attributes only have an effect on route selection. These include `LOCAL_PREF`, `MULTI_EXIT_DISC`, `ORIGIN` and `AS_PATH`, plus extensions such as

[1] Timer randomisation and triggered updates make this worst case very unlikely to occur in practice, but can't guarantee it will never happen.

communities. The main exception is the NEXT_HOP attribute, which is used to set up forwarding tables.

As with gRIP, we have developed a modified version of BGP to allow new routing languages to be used in tightly-defined extension points with no further changes to the algorithm. Our design for gBGP replaces all of the route-selection attributes with a single attribute that is controlled by the chosen routing language. Routes must have the NEXT_HOP and new METAROUTING_METRIC attributes, and no others. As with gRIP, the routing language must serialise metric values to and from a variable-length byte stream, and the algorithm does not care about the structure of those bytes. BGP's decision process is replaced with the routing language's metric comparison operation. As BGP assumes the decision process will only return a single best route, the routing language must provide a total order.

One significant design choice here is that we consider the AS_PATH attribute to be part of the language, not part of the algorithm. gBGP therefore does not provide the loop-freedom guarantees that AS paths provide – as with RIP, it is up to the language to deal with this (perhaps by including a mechanism equivalent to AS paths or perhaps by taking a different approach). Unlike RIP, the BGP algorithm is designed with the assumption that the AS path will prevent any loops that may lead to counting-to-infinity states. Whereas RIP will propagate stale route information with a worst case of 30 seconds per hop, and a typical case that is much faster, BGP has a best case of a configured constant "MinRouteAdvertisementIntervalTimer" (recommended as 30 seconds by the RFC) per cycle, as an enforced delay to intentionally rate-limit oscillations. Further, BGP route flap damping [VCG98] can introduce far more extreme rate-limiting – a prefix receives a penalty each time the router sees it change, slowing and eventually stopping the propagation of routing information, and the penalty decays exponentially with a half-life of typically 15 minutes [Cis09a]. A RIP-like routing algebra that relies on counting-to-infinity with a small infinity will therefore have disastrously bad convergence performance in BGP, even though the algebraic properties show it will eventually converge correctly.

An alternative design choice would be to retain the AS_PATH attribute and consider it an inherent part of the algorithm, given how fundamental it is in the design of BGP. We will call this algorithm gBGP_{AS}, and it will guarantee loop-freedom for any routing language. Although BGP uses the AS_PATH length as a factor in its decision process, it comes after comparing LOCAL_PREF and before comparing attributes such as MULTI_EXIT_DISC. Since we still want gBGP_{AS} to be reasonably generic, the local preference and MED attributes will be part of the routing language and not hardcoded in the algorithm. The algorithm can only use the routing language's metric comparison as an indivisible operation, and cannot slip an AS_PATH length comparison into the middle of two of the routing language's fields (as would be necessary to emulate BGP's behaviour) without violating the language/algorithm abstraction. We must therefore

say that AS_PATH is used only for detecting loops, and its length is not used as part of the decision process. Instead the routing language must perform its own path computation if it wishes to use path lengths in its order. This separation of loop-detection from length-determination introduces some redundancy into the protocol, but provides a cleaner separation of responsibilities.

BGP's capabilities mechanism is used to ensure gBGP neighbours are using the same routing language. When setting up the connection, the routers compare a 16-byte unique ID that identifies the routing language and refuse the connection if it is inconsistent. When the routing language is generated by the ERL-to-C++ compiler, this unique ID is a cryptographic hash of the parsed ERL syntax tree. This allows the use of any number of languages with negligible chance of accidental ID collisions, and without the overhead of using an external numbering authority.

4.3 Generalising other protocols

This dissertation focuses on RIP and BGP, but it is useful to explore how well the language/algorithm split can apply to other vector routing protocols based on the Bellman-Ford algorithm.

4.3.1 EIGRP

EIGRP [AGLAB94] is a distance-vector protocol which explicitly defines a special composite metric type with addition and comparison operations, corresponding to our notion of a routing language, and an algorithm that uses that language.

To prevent routing loops at any time during convergence (and therefore to prevent counting-to-infinity), EIGRP uses DUAL (the Diffusing Update Algorithm) [GLA93]. This is designed to allow fast convergence, while preventing a router from selecting a route that could potentially loop back to itself. The only assumption DUAL makes about the metric is that it is increasing: if a router sees a route to a destination with a more preferred metric than it has recently advertised itself for that destination, then it knows the route cannot pass through itself and is safe to use immediately. If it cannot find any unambiguously safe routes, it switches to a more expensive convergence mechanism (the *diffusing computation*; effectively a variation of the distributed Bellman-Ford algorithm where a node asks its neighbours to update their own routes (which may require them to perform a diffusing computation themselves) before using their newly updated state in its own step of the DBF algorithm, to prevent transient loops caused by the use of stale information).

The EIGRP metric consists of four integer components: bandwidth (stored in an inverted form so smaller numbers are better), delay, load and reliability. Each component

is processed separately in route advertisements: delays are combined with addition, inverse-bandwidths and loads with max, and reliabilities with min. When comparing routes, the components are arithmetically combined into a single integer with user-configurable weights (which must be consistent across the network).

The routing language can be expressed in our algebraic model as:

$$\begin{aligned} S &= \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\ L &= \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\ (a_B, a_D, a_L, a_R) \otimes (b_B, b_D, b_L, b_R) &= (\max(a_B,\ b_B),\ a_D + b_D,\ \max(a_L,\ b_L),\ \min(a_R,\ b_R)) \\ a \preceq b &= W(a) \leq W(b) \end{aligned}$$

where the comparison operator maps each composite metric onto a scalar number with the weight function W,

$$W(a_B,\ a_D,\ a_L,\ a_R) = \begin{cases} \left((K_1 a_B + \frac{K_2 a_B}{256 - a_L} + K_3 a_D) \cdot \frac{K_5}{K_4 + a_R} \right) \cdot 256 & \text{if } K_5 \neq 0 \\ \left((K_1 a_B + \frac{K_2 a_B}{256 - a_L} + K_3 a_D) \right) \cdot 256 & \text{otherwise.} \end{cases}$$

As long as the constant weights K are positive, this is increasing, but for most combinations of K it is not distributive.

In practice, the link label's load and reliability values (a_L, a_R) are calculated dynamically by the router software based on measurements of relevant factors, so they are likely to fluctuate over time. Conceptually this is the same as manual reconfiguration of policy on a router, but may be much more frequent and may have consequences on the algorithm design to ensure efficient performance. However, Cisco strongly suggests setting $K_2 = 0$ and $K_5 = 0$ so that only bandwidth and delay are used for route selection: "Although you can configure other metrics, we do not recommend it, as it can cause routing loops in your network." [Cis04]; the DUAL algorithm provably prohibits loops when used as designed, so it may be assumed that Cisco's EIGRP does not exactly match DUAL, most likely by failing to handle dynamic changes to load/reliability with a full recomputation (which may trigger an expensive diffusing computation) and instead using a simpler update process which may violate the feasibility conditions defined by DUAL, though this is speculation given the lack of detailed documentation. Whatever the reason, the lack of use of these dynamic attributes indicates they are more trouble than they are worth, so we will not explore them further.

Although we can express this routing language as an algebra, there is no way to express the additive composition of values in the current version of RAML. However, this could likely be added as a future extension of RAML without any fundamental changes. Just as we can combine two algebras with a direct product or a lexicographic product (which differ only in their definition of $\preceq$), we could imagine combining two algebras $(S, L_1, \preceq_S, \triangleright_1)$ and $(S, L_2, \preceq_S, \triangleright_2)$ with an additional order semigroup

$(S, \preceq_S, \otimes)$ to give a new algebra similar to a direct or lexicographic product but with

$$(a_1, a_2) \preceq (b_1, b_2) \Leftrightarrow (a_1 \otimes a_2) \preceq_S (b_1 \otimes b_2).$$

To model the simplified bandwidth-delay EIGRP metric with this, we would combine bandwidth $(\mathbb{N}, \mathbb{N}, <, \max)$ and delay $(\mathbb{N}, \mathbb{N}, <, +)$ with the order semigroup $(\mathbb{N}, <, +)$ to produce an algebra that compares the sum of bandwidth and delay. The main complication is determining the property inference rules: in this case the three input algebras are non-decreasing and distributive, and the resulting algebra is increasing but non-distributive, but a comprehensive analysis would be necessary for RAML to support inference of properties for arbitrary algebras. We do not attempt this work here.

Gouda and Schneider [GS03] define the term "maximizable routing metric" to refer to what we would call routing algebras that are non-decreasing ("bounded" in their terms) and distributive ("monotonic"), and show those properties are necessary and sufficient for a distance vector algorithm to find a global optimum solution ("maximum metric tree"). They find that a simplified EIGRP metric with $W(a_B, a_D) = a_B + a_D$ is non-distributive, and further that non-distributivity violates some assumptions about DUAL, resulting in poor performance characteristics due to the potential for an update to unexpectedly trigger more than a single diffusing computation. However, they do not explore the meaning of the routing solution that the algorithm will find, which we now understand as a local optimum.

Given this separation of EIGRP metric language and algorithm, we can imagine a gEIGRP protocol along the same lines as gRIP, replacing the hard-coded metric value with an arbitrary variable-length one determined by the routing language. Unlike BGP AS paths, the DUAL loop prevention approach is clearly a part of the algorithm and not a part of the language – it is tightly integrated with details of network packet processing, and the algorithm must maintain various extra tables in order to implement it. Unlike gRIP, a routing language used with gEIGRP does not have to deal with the counting-to-infinity problem as the algorithm itself will prevent such a situation.

4.3.2 AODV

The AODV protocol [PBRD03] uses sequence numbers to prevent loops, and an 8-bit hop count as the metric. The metric is used as a *feasibility condition* – new routes with an equal sequence number to the current route selection are accepted if the new route's hop count is less than or equal to the current route's, since they couldn't possibly be part of a cycle through that router. This simply requires the metric to be increasing, so it could come from any increasing routing language. However, the hop count is also used to initialise IP TTLs for route request flooding, so it straddles our language/algorithm

divide and there is not as clean a separation as in the other protocols we have examined. If we were to design a gAODV protocol where the routing language metric is opaque from the algorithm's perspective, we would have to compute the hop count alongside the metric so that the algorithm can continue to use it for the flooding process. However, the use of complex metrics would go against AODV's goals of low processing and memory overhead and network utilisation, so gAODV would likely be unsuitable for the environments for which AODV was designed while still using an algorithm designed for those tight constraints.

4.3.3 Babel

The Babel routing protocol [Chr11] uses sequence numbers similarly to AODV, with the goal of performing better in unstable networks than protocols such as RIP. The specification explicitly discusses the metric using the terminology of Sobrinho's routing algebras:

> the function $M(c, m)$ used for computing a metric from a locally computed link cost and the metric advertised by a neighbour MUST only satisfy the following conditions:
>
> 1. if c is infinite, then $M(c, m)$ is infinite;
> 2. M is strictly monotonic: $M(c, m) > m$.
>
> Additionally, the metric SHOULD satisfy the following condition:
>
> 1. M is isotonic: if $m \leq m'$, then $M(c, m) \leq M(c, m')$.

However, the Babel algorithm assumes metrics are positive integers and can be compared with $<$ and encoded in 16 bits with 0xFFFF for infinity – the genericity is solely in the computation of a new integer metric, based on the old integer metric and any extra node state (e.g. battery status) and link cost (e.g. estimated from packet loss stats), while the metric type and comparison are hard-coded into the algorithm, providing very limited flexibility. Fully generalising the algorithm using metarouting's routing languages would be a similar process to developing gRIP, replacing the algorithm's 16-bit metric fields with a variable-length metric and using the language's comparison operation instead of integer less-than. One complication is that the Babel RFC states "A node SHOULD NOT send triggered updates [...] when there is a minor fluctuation in a route's metric" – it does not define a minor fluctuation, but the intent is to reduce network traffic if the metrics depend on random variables such as battery status whose changes are usually insignificant. Our current definition of routing languages has no support for the concept of distinguishing minor fluctuations, though nor does

it have the concept of depending on fluctuating external variables; a version of gBabel using the current metarouting system would simply ignore these complications and work correctly without them, though they suggest future possibilities for extending our notion of routing languages.

4.4 Differences with the model

In this description of protocols, we have made a few simplifications in order to match the algebraic routing model described earlier.

One of these simplifications is the concept of a network configuration as a set of labelled edges between nodes. In most protocols, these labels do not exist as explicit data. Instead, each router has its own configuration data that controls both how it *imports* routes from the network, and how it *exports* routes back to the network.

For example, by default a RIP router increments the metric when exporting a route, but in typical implementations it can be configured to update the metric on both import and export. This does not add any expressive power to the protocol (it is always equivalent to some graph whose arc labels are the sum of import and export increments), but it may simplify the configuration of certain network designs. BGP defaults to performing some computation on import (such as setting `LOCAL_PREF`) and some on export (such as setting `MULTI_EXIT_DISC`) – in this case a router may be talking to a router that is under separate administrative control, and the location of the configuration data and of the processing is critical for security and information-hiding purposes.

We discuss this issue in detail in the Chapter 5. For this chapter, we assume that all configuration and processing occurs on export, and routes are imported from the network directly into routing tables with no further modification. This has only a minor effect on the implementation concerns discussed here, so we ignore the extra complication for now.

Another issue is the origination of routes. These typically come into a protocol via *redistribution* from either another dynamic protocol, or from a statically-configured set of routes. Redistribution is a complex topic and we do not discuss it here – we simply assume the existence of some mechanism to inject routes with valid metrics into the protocol. Some related work has begun to explore redistribution [LXZ07, LXP+08, LXZ10] and its role in the metarouting system [BG09, Ali11].

We are focusing entirely on routing, and ignoring the distinct concept of forwarding. We assume a direct correspondence between the paths of route advertisements and of data packets, whereas protocols such as BGP can set up routes based on different next-hop addresses (especially with route reflectors, which partially centralise the routing

computations without centralising the forwarding), and redistribution between multiple protocols can result in forwarding tables that do not match any individual protocol. While these are important features for practical routing implementations, they are not part of the core functionality of routing, so we will continue assuming routing and forwarding are equivalent.

4.5 Implementing gRIP and gBGP

We have shown how current protocol designs can be split into language and algorithm components, which allows us to reason about protocols using the algebraic model. The question this section addresses is whether we can carry this conceptual split through to current protocol implementations, and use them as implementations of gRIP and gBGP that tie into routing languages produced by our metarouting compiler.

We used the RIP and BGP implementations of the Quagga routing suite for this work. Quagga is a fairly well-established open source system, written in the C language. Section 2.4 gives a brief overview of its design. We use the term gQuagga to refer collectively to our generalised algorithm implementations.

4.5.1 Linking

Given a gQuagga algorithm and an executable routing language produced by the metarouting compiler, we need a way to link the pieces of code together. There is a range of feasible options, from very tightly coupled to very loosely:

Source insertion – place the language implementation code directly into the relevant locations in the algorithm, either by editing the source files or using C preprocessor macros, and then compile the combined code.

Static linking – compile the language and algorithm implementations into object files independently, with the algorithm calling functions defined by the language, and then use the C compiler's linker to combine them into a single executable.

Dynamic linking – compile the language into a dynamically-loaded shared library, with the algorithm compiled into an executable that loads the library at run-time and calls into it using function pointers.

Inter-process communication (IPC) – compile the language and algorithm into separate executables, that can connect and communicate at run-time using a standard IPC mechanism such as sockets.

Recall that the metarouting compiler emits C++ code. This makes source insertion difficult, as Quagga is written in C – its code and build system would have to be ported to C++, which is a large and disruptive change. (The problem would be avoided if the metarouting compiler emitted plain C code, but there were compelling reasons for it to choose C++ instead.) Static linking avoids this difficulty, as object files produced by the C compiler and C++ compiler can be linked together if they define compatible interfaces. However, this requires a function call for every operation on a metric (whereas the compiler could easily inline and optimise functions that were inserted directly into the algorithm code), and imposes a generic interface between the algorithm and language that may not be optimal in all cases (e.g. requiring explicit "copy" and "free" operations on metrics, in case they must perform memory allocation, even if the particular chosen language does not need this), which may have a measurable performance impact on code that performs large amounts of metric processing.

Both of these options require that the protocol executable is recompiled or at least relinked whenever the language is changed. This is a fairly slow process, and requires a means to distribute the new executable to the routers that will run it. Dynamic linking avoids this by performing the linking at run-time, so only the language itself (which is typically very small) needs to be recompiled and distributed when it changes. This requires more complex code in the routing algorithm, to load the library and initialise function pointers to call into it, and the indirection adds a small run-time cost to each function call. It also requires operating system support for dynamic loading which may be unavailable on some non-general-purpose routing devices.

IPC imposes the least restrictions – the algorithm and language do not even have to be written in languages that have binary-compatible interfaces (as C and subsets of C++ have). By using TCP sockets for communication, they do not even have to be running on the same physical machine. Quagga already uses an IPC mechanism, with each routing protocol running in its own process and asynchronously communicating with the central "zebra" daemon (which coordinates between the protocols and updates the router's forwarding tables) over sockets that can be either local or networked. Unfortunately this approach has high latency, making it unsuitable for the frequent synchronous operations that a routing algorithm requires.

As a result of these considerations, and given the benefits of rapid iteration to our early experimental system, we chose to use dynamic linking for gQuagga. A stable production-quality system running on embedded hardware may make a different choice if it chose to separate the algorithm and language in this way.

There are two additional layers between the algorithm and language. Since the metarouting compiler generates C++ code using features like classes and templates, we need a wrapper that converts it to a C-compatible API of functions and pointers. This is the *mrc API*. Additionally we have a layer inside gQuagga that deals with the com-

plexities of loading the library and setting up function pointers, and also provides a type-safe API (by wrapping pointers in `structs`) and various debugging features (memory leak detection, error reporting) that are not provided by the routing language implementation itself. The rest of the gQuagga code calls functions provided by this internal API, keeping all the implementation details hidden from the gRIP and gBGP algorithm code.

To illustrate the details of this API layering, we will consider the example of gRIP performing a route comparison. gRIP contains code such as

```
if (mrc_metric_is_better (routing_language, metric, rinfo->metric))
{
    ...
}
```

where `routing_language` was previously loaded by a call to `mrc_init("filename.so")` and contains global state (primarily a pointer to the shared library that implements the routing language). Here `metric` is a metric that was decoded from a network packet, and `rinfo->metric` is the metric of the previous best route to that destination.

`mrc_metric_is_better` is part of the mrc API, shared between all gQuagga routing algorithms. It is implemented as

```
int mrc_metric_is_better(mrc_language_t lang, mrc_metric_t x, mrc_metric_t y)
{
    ASSERT_VALID_METRIC(x);
    ASSERT_VALID_METRIC(y);
    return DLL(lang)->metric_is_better(SIGVAL(x), SIGVAL(y));
}
```

When compiled in debug mode, `ASSERT_VALID_METRIC` checks the value is non-null and of the correct type and that its memory has not been deallocated, to help detect bugs and misuses of the API that would otherwise cause incorrect behaviour. This is necessary because of C's lack of strong type-safety and automatic memory management. The type `mrc_metric_t` is based on the structure

```
typedef struct sig
{
    int freed;
    const char *source;
    void *backtrace[16];
    sig_type type;
    struct language *lang;
```

```
    void *value;
} sig;
```

where the first five fields are for tracking allocation details for error detection and error reporting (omitted when not compiled in debug mode), while the `value` field is a pointer to an object managed by the shared library implementing the routing language. `mrc_metric_is_better` uses the `SIGVAL` macro to extract this `value`, and passes it to the function pointer loaded from the shared library.

The function implementation in the shared library is

```
DLLEXPORT int mrc_impl_metric_is_better(void *m0, void *m1)
{
    return plus()(*(metric_t *)m0, *(metric_t *)m1) == MRC_ORD_LT;
}
```

This function is implementing the mrc API. Everything before this point has been part of gQuagga, but this is the standard interface between routing languages and routing algorithms – the same `mrc_impl_metric_is_better` could be called by a non-Quagga-based routing algorithm. (For example, we have a Python-based network simulator which uses this API to help test the behaviour of a routing language.)

This function is again a wrapper: the real functionality is provided by the C++ code which the mrc compiler generates from an ERL specification. For example, `metric_t` and `plus` may be defined as

```
typedef AddConst<IntBound<0, 15>, str_INF> metric_t;
typedef AddConstTop<IntLte> plus;
```

to provide a RIP-like routing language. This generated code is compiled against *libmrc*, which provides the standard definitions of `AddConst`, `IntBound`, `AddConstTop` and `IntLte`, and against the mrc API wrapper which defines the `mrc_impl_metric_is_better` function in terms of the C++ template types.

4.5.2 Configuration syntax

A critical part of any routing protocol implementation is its configuration interface. Quagga borrows heavily from Cisco's configuration syntax. In this section we will discuss how a routing language produced by the metarouting compiler is integrated into this interface. Other protocol implementations have very different syntax, but similar concepts are applicable.

```
router bgp 1111
  bgp router-id 10.0.1.4
  neighbor 10.0.2.2 remote-as 2222
  neighbor 10.0.2.2 route-map EXAMPLE1 in
  neighbor 10.0.1.3 remote-as 1111
  neighbor 10.0.1.3 route-map EXAMPLE2 out

route-map EXAMPLE1 permit 5
  match as-path 2100
  set local-preference 100

route-map EXAMPLE2 permit 5
  set metric 100
```

Figure 4.1: BGP configuration file example.

Figure 4.1 gives an example of Quagga's standard BGP configuration syntax. This sets up the router to run BGP with AS number 1111. It initiates connections to two neighbouring BGP routers, the first in a different AS (2222) and the second in the same AS. It sets up a different route map for each neighbour to define its routing policy.

Configuration commands can be put into a file, or entered into an interactive terminal. In the interactive mode, the tab key is used to auto-complete commands and provides documentation about all possible choices.

Quagga implements its configuration interface with a series of command elements. Each of these has a template string that defines the command's keywords and variables, plus a corresponding function that is called when that command is entered. A template like `"set dist <1-255>"` will accept commands such as `"set dist 10"`, restricted to the given integer range, and a template like `"set name STRING"` will accept any token for the variable. Each line of the user's input is matched against these patterns; if none match then an error is reported, else the corresponding function is executed with the template's variables passed as arguments.

The command lists are constructed by Quagga at run-time (by very large amounts of verbose C code), so it is possible for us to add extra commands during the dynamic loading of routing languages. Other protocol implementations such as XORP construct code for their configuration interface at compile-time based on a declarative specification of the configuration commands, which may make them trickier to extend at run-time in this way and may require a different approach.

Figure 4.2 sets up a gBGP router. This is a demonstration of how the use of a declarative language like ERL allows us to automatically generate boilerplate code necessary for running a routing language as part of a complete protocol implementation. The list

```
routing-language scoped_prod

router bgp
  bgp router-id 10.0.1.4
  neighbor 10.0.2.2 route-policy EXAMPLE1 out
  neighbor 10.0.1.3 route-policy EXAMPLE2 out

route-policy EXAMPLE1
  set external ecomm 2
  set external epath 2222
  set external idist 0
  set external ipath empty
end-policy

route-policy EXAMPLE2
  set internal idist 1
  set internal ipath 1111
end-policy
```

Figure 4.2: gBGP configuration file example.

Expression e	Configuration commands $[\![e]\!]$
`TyUnit`	$\{$`"unit"`$\}$
`TyInt`	$\{$`"<-2147483648-2147483647>"`$\}$
`TyIntNonNeg`	$\{$`"<0-2147483647>"`$\}$
`TyIntPos`	$\{$`"<1-2147483647>"`$\}$
`TyIntRange(`n`, `m`)`	$\{$`"<`n`-`m`>"`$\}$
`TyString`	$\{$`"STRING"`$\}$
`TyList(`t`)`	$\{$`"empty"`$\} \cup \{$`"INDEX `e`"` $\mid e \in [\![t]\!]\}$
`TyListSimp(`t`)`	$\{$`"empty"`$\} \cup \{$`"INDEX `e`"` $\mid e \in [\![t]\!]\}$
`TySet(`t`)`	$\{$`"empty"`$\} \cup \{$`"INDEX `e`"` $\mid e \in [\![t]\!]\}$
`TySetMin(`t`)`	$\{$`"empty"`$\} \cup \{$`"INDEX `e`"` $\mid e \in [\![t]\!]\}$
`TyAddConst(`c`, `t`)`	$\{\text{lowercase}(\texttt{"}c\texttt{"})\} \cup [\![t]\!]$
`TyRecord(`n_1`=`t_1`, `n_2`=`t_2`)`	$\{\texttt{"}n_1\ e\texttt{"} \mid e \in [\![t_1]\!]\} \cup \{\texttt{"}n_2\ e\texttt{"} \mid e \in [\![t_2]\!]\}$
`TyUnion(`n_1`=`t_1`, `n_2`=`t_2`)`	$\{\texttt{"}n_1\ e\texttt{"} \mid e \in [\![t_1]\!]\} \cup \{\texttt{"}n_2\ e\texttt{"} \mid e \in [\![t_2]\!]\}$

Figure 4.3: Extraction of configuration commands from ERL type syntax.

of `set` commands is generated recursively from the tree structure of the ERL expression defining policy types. To match the traditional design of Cisco-like configuration languages, we choose to flatten the tree structure into a flat list of commands that each specify the value at a leaf node. Although a hand-crafted configuration syntax has the potential to provide a more natural and convenient interface, this automatic approach can handle any arbitrary ERL type while being reasonably elegant for the common cases.

Based on the ERL syntax in Chapter 3, we define the configuration commands generated for any ERL type expression in Figure 4.3. For example, the type

```
TyRecord(a = TySet(TyAddConst(INF, TyIntNonNeg)), b = TyUnit)
```

will expand into the following set of commands:

```
a empty
a INDEX inf
a INDEX <0-2147483647>
b unit
```

The value $(\{100, 200, \infty\}, 1)$ could then be specified in the configuration file as

```
set a 0 100
set a 1 200
set a 2 inf
set b unit
```

This handling of lists and sets with explicit indexes and a single command per element is somewhat inelegant, but it permits us to have complex record types inside a list without the syntax becoming unbearable. It would be possible to add an optimisation for the common case of lists in which each element is specified as a single token, so that the user can specify the entire list in a single line of space-separated tokens, while retaining the currently-defined approach for scalability to more complex types, but we do not do this here.

Along with generating the list of commands, we also generate code that will either return an error message or construct a value of that type, given a set of commands entered in the configuration file. Errors include failing to specify values for required leaf nodes in the type expression tree, or specifying values for more than one branch of a disjoint union. There is also code to convert a value back into a list of commands that will construct it, so that the router can print or save its configuration state.

This translation from ERL types into code that sets up the Quagga configuration system is written entirely in C++, using partial template specialisation over the templated

representation of ERL types that the mrc compiler generates. It is then exposed through the mrc API so that it can be accessed by gQuagga. This means the mrc API can continue to treat the routing language's types and values as opaque objects, and there is no need for a "reflection" ability to expose the structure of these types that would let gQuagga construct configuration commands by itself. However, this does mean that mrc must be aware of some of the implementation details of Quagga in order to construct the function pointers and data structures that are exposed through the API, which slightly blurs the abstraction between language and algorithm.

Although our implementation is only for Quagga's configuration syntax, the same concepts can apply to configuration of XORP or to JunOS-like syntax. This automatic approach to configuration demonstrates a significant benefit of the metarouting system: by restricting ourselves to the set of routing languages that can be expressed in ERL, and by specifying routing languages declaratively, we can automatically provide a clean, consistent, complete integration with these low-level details of the routing algorithm, freeing the language designer from the need to manually implement all the details before they can start experimenting with a new routing language.

4.5.3 Drawbacks of generalisation

Implementing these generalised routing algorithms using Quagga reveals a few difficulties with the details of the approach. The most problematic change to the protocol semantics is that RIP has a very simple wire protocol, and the gRIP extensions lose some of that simplicity due to the use of variable-length metrics. This particularly affects the behaviour of non-whole-table Request messages. In standard RIP [Mal98], these consist of a single UDP packet with one fixed-size "route entry" (RTE) per destination for which they wish to request the route metric. The RTE metric fields are left blank in the request. A router that receives such a packet simply fills in the metric fields of each RTE entry, flips the 'command' byte in the header to indicate a response message, and sends the packet back. In gRIP, the use of variable-length metrics means that the request message cannot pre-allocate blank space for the metric fields, and the response message may have to be split into many UDP packets. This entails greater complexity and processing cost on the responding router, and since UDP is an unreliable protocol the requesting router may receive response packets for only a part of its request. In this case, the generalised protocol sacrifices some of the relative elegance in the original protocol's design. In contrast, BGP is already designed for extensibility and variable-length route attributes, so the transition to gBGP does not conflict with the standard wire protocol's design.

Code complexity is a danger when modifying software in ways it was not originally designed to handle, but our experience with Quagga is that this has not been a signif-

icant problem. The abstraction provided by the mrc API is a close match to the operations performed by the standard Quagga code, so most of the code could be ported with very localised changes. The most widespread change was to perform explicit copy/free of metric values, as variable-length metrics must be allocated in heap memory instead of on the C stack, and C provides no facilities for automatically managing memory; the debugging facilities added by the gQuagga wrapper around the raw mrc API were valuable in detecting problems with this. The largest chunk of new code is for integrating the configuration syntax described earlier, but in this case it is replacing a large amount of hand-written protocol-specific configuration command code – the result is a slight increase in conceptual complexity but also a simplification by reducing the amount of code. In conclusion, the careful design of abstractions and interfaces allows a protocol implementation to be generalised to support externally-specified routing language with no major complexity cost.

4.5.4 Performance

One further concern is the performance impact of the changes, particularly the increased overhead from the indirection of metric computations, and the loss of protocol-specific optimisations (such as Quagga's BGP implementation using a hashing scheme to share common data such as AS paths between multiple routes). This section describes an initial investigation into the performance of the gQuagga implementation, aiming first to quantify the inherent cost of the abstraction, and second to indicate the extent to which the complexity of the routing language affects the overall performance of the complete protocol.

Note that we are only interested in measuring control plane performance, not data plane performance; our modifications do not affect the forwarding behaviour of the router. Control plane performance is still an important issue as it affects the convergence time after a topology change.

We performed all measurements using the gBGP algorithm; gRIP is less suitable for comparison as the RIP protocol and implementation are designed for small networks where CPU cost will be negligible, whereas BGP scales to network sizes where performance is important. To provide a relatively fair comparison between standard BGP and the generalised version, we modelled a part of BGP's AS path functionality in RAML, and compared against the unmodified BGP protocol by sending route update messages that used only AS paths, so that most other parts of BGP's route selection mechanism were skipped. We also ran a version of the scoped product example from Section 3.3.3 to see the effect on performance of a more complex routing language.

Methodology

All experiments were performed with three routing protocols:

- `std` – the original BGP implementation from Quagga. The test devices sent UPDATE messages containing only the mandatory `AS_PATH`, `ORIGIN` and `NEXT_HOP` attributes.
- `paths` – our implementation of gBGP, heavily based on Quagga's BGP with minimal modifications. This uses a RAML specification representing lists of integers bounded between 0 and 65535: `paths(int_range_min_plus(0, 65535))`.

  The tests sent UPDATE messages containing only the `NEXT_HOP` and our new `METAROUTING_METRIC` attribute.
- `regions` – the same gBGP implementation but using a variation of the scoped product example described earlier, with the path fields changed to lists of strings instead of lists of integers.

The source data came from public Internet routing table dumps[2] containing about 250,000 distinct prefixes. This large amount of data is helpful for investigating performance as it provides a realistic distribution of prefixes and path lengths.

CPU time and dynamic memory usage measurements were taken with the `getrusage` and `mallinfo` library calls from within Quagga's BGP process. This means the time measurements ignored periods when the process was inactive, such as while waiting for various timers, and do not give an accurate picture of how long the routing protocol will take to converge. As metarouting is changing only the routing language, not the algorithm, these timer delays are outside the area we are interested in, and ignoring them allows a much more precise view of the relative routing language performance.

Our test setup consisted of a desktop machine with an Intel Core 2 Quad CPU with frequency scaling locked at 2.4GHz and 4GB RAM, running Quagga 0.99.9 on x86_64 Linux 2.6.23. All code was compiled in 32-bit mode to reduce memory usage.

Test devices were written using the Net::BGP Perl module for session maintenance, with the UPDATE messages precomputed and written directly to the TCP socket to minimise the runtime delay. All processes were run on a single machine, to avoid the effects of network latency and bandwidth limitations. The Quagga BGP process was configured to not update the kernel forwarding table, as we were only interested in the routing protocol itself.

For the first experiment, we ran the routing process with an initially empty routing table, then sent a number of UPDATE messages (one per distinct prefix) from a single

[2] `http://data.ris.ripe.net/rrc01/2008.07/bview.20080702.0759.gz`

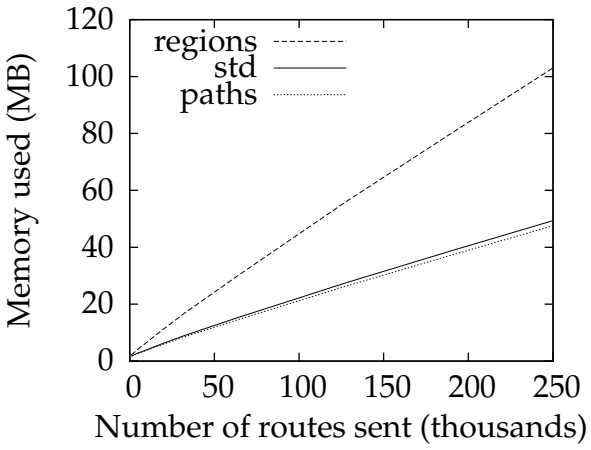


Figure 4.4: Dynamic memory usage of BGP and gBGP routers against number of received routes.

test device connected and configured as a neighbour. This let us measure the memory used to store the routing table, and the processing time required to read metrics from the network and apply import policy.

We ran two further experiments to highlight different aspects of the CPU time cost, each with two test devices connected to the routing process. In the first of these, only one test device sent UPDATE messages, and the CPU time was recorded after these routes had been propagated to the second test device. In the second case, both devices sent UPDATE messages (with the same prefixes but slightly different AS paths) so that the router would have to compare two routes for each prefix to pick the best, and the processing time was recorded before the new best routes had been propagated to the test devices.

Results

Figure 4.4 shows the memory usage against the number of UPDATE messages in the first experiment. The memory used before storing any routes is similar in each protocol (around 1.5MB), and the per-route memory is the significant factor. (These measurements exclude static memory, such as constant data and code. The gBGP routing language libraries add roughly 100KB to this value, compared to the original BGP implementation.)

The `paths` case required about 4 bytes less memory per routing table entry than `std`. This is because Quagga's BGP implementation stored AS paths as a linked list of sets of integers to handle the aggregation and confederation features of BGP, whereas our routing language omitted these details and was compiled into a dynamically-sized array of integers.

Test	`std`	`paths`	`regions`
receiving	4.2 (100%)	4.5 (107%)	8.2 (195%)
sending	5.8 (100%)	6.2 (107%)	11.0 (190%)
comparing	5.5 (100%)	6.2 (113%)	11.3 (205%)

Figure 4.5: CPU time in seconds (% relative to `std`) for receiving 250K routes from a neighbour; receiving 250K routes then sending to a second neighbour; and receiving 125K routes from each of two neighbours and comparing to select the best. Median of three runs. Standard deviation is below 3%.

Figure 4.4 also shows the more complex `regions` example, where each metric includes distance fields and the paths consist of strings instead of integers. This doubles the total amount of memory used – the metric is a significant part of the overall cost of each route.

Figure 4.5 shows the CPU time used by the BGP/gBGP process in each experiment. The `paths` example is roughly 5–10% slower than `std` for all operations.

The `regions` example takes nearly twice as much time as `std`: the increased complexity of the metric and policy means all operations are necessarily slower, and as with memory usage this has a significant effect on the overall route computation performance.

For a fair comparison against standard BGP we would have to implement the full BGP feature set in a RAML specification, but these initial results demonstrate that our generalisation of the routing algorithm does not have a major performance cost. The cost is instead a factor of the routing language, and our results show this can be significant in our current implementation. Billings [Bil09] explores various approaches to optimising the compiled C++ implementation of routing languages, including carefully choosing efficient data structures and sharing common sub-values between metrics, as well as more specialised optimisations based on the automatically-computed algebraic properties of the routing language.

In summary, we do have to make some tradeoffs in performance and complexity to support our generalised model of routing protocols, but the problems are fairly minor and we consider it an acceptable cost for the benefits of the added power and flexibility provided by the generalised protocol implementations.

Chapter 5

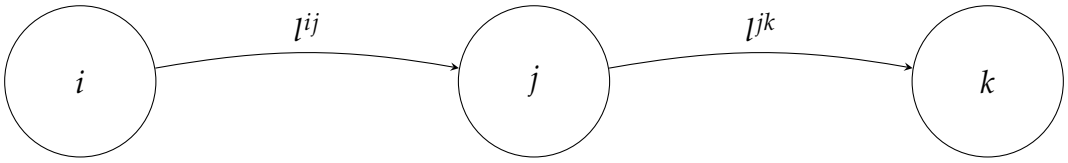
Configuration

5.1 Arc configuration vs. interface configuration

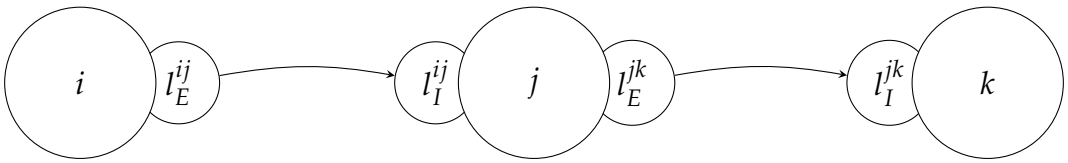
Until this point, we have used the standard graph model of a network when modelling the behaviour of a routing protocol. We view a link between two routers as a labelled arc between nodes, where the label represents the routing policy, as in Figure 5.1a. As described in Section 2.1, a network configuration is determined entirely by the graph layout (V, E) and the weight function $w : E \to L$. The edges and weights can also be encoded in an adjacency matrix $\mathbf{A}$.

In practice, this is an overly simplified view of vector routing protocols. Link-state protocols may explicitly construct a representation of the network as an adjacency matrix based on link state advertisements received from other routers, but in vector protocols the network-wide behaviour is an implicit consequence of purely local decisions in the distributed algorithm, and these local decisions do not closely match the model we have been using. In particular, in a vector protocol implementation there is no single obvious place to store a representation of an arc weight or to apply it to a routing metric. The physical link has no storage or computational ability itself, so this must instead be handled by one of the two routers connected to the link – but which one?

In the distributed Bellman-Ford algorithm described in Section 2.1.2, router v knows the adjacency matrix entry $\mathbf{A}(u, v)$ and uses that value in its computation. However, in all common vector protocols the link policy is split across *both* routers. The first router has some policy specified in its configuration file, which it applies to a metric before *exporting* it from its routing table onto the network link. (If it has links to many routers, it may have a different export policy for each link.) The second router has an independent policy in its own configuration file, which it applies to metrics received over the link before *importing* into its routing table. (Similarly it may have a different import policy per link.) This complication is ignored in our original algebraic model of routing, but in this section we argue it is an important concern that should be examined



(a) Arc configuration.



(b) Interface configuration.

Figure 5.1: Associating policy labels with (a) arcs, or (b) interfaces between nodes and arcs.

through an extension of the algebraic tools.

Figure 5.1b illustrates how policy labels are no longer associated with arcs; instead they are associated with the *interfaces* between nodes and arcs. In place of labels $l \in L$, we have two separate label types $l_E \in L_E$ and $l_I \in L_I$.

The gap between the arc-based algebraic model and the interface-based vector protocol implementation is a significant problem for the metarouting system as described so far. The algebraic correctness properties will only have practical relevance if the implementation closely matches the algebraic model – a large gap provides many opportunities for unexpected un-modelled behaviour that breaks the protocol's correctness. It is also a challenge for metarouting as a mechanism for specifying implementations: if many implementation details (such as the split between export and import policy) are hard-coded into the routing algorithm implementation, the flexibility of routing languages is greatly reduced. We therefore want a way to bridge the gap between algebra and implementation. This chapter aims to build a section of the bridge to handle interface-based configuration in the metarouting system. The design is an extension of work published earlier by the author [TG09].

5.1.1 Benefits of import and export policy

There are two easy ways we could change a vector routing protocol to better match our original arc configuration model: put the policy data and computation entirely on the exporting router, or put it entirely on the importing router, so there is a direct correspondence between policy and arc labels. The policy-on-export case is what we have assumed throughout the previous chapter. However, there are important practical advantages to supporting separate policy on both import and export interfaces and changing the model to match this.

Some of these advantages are only applicable when some neighbouring routers are under separate administrative control, and will not be important concerns in smaller homogeneous networks, but some apply equally at all scales.

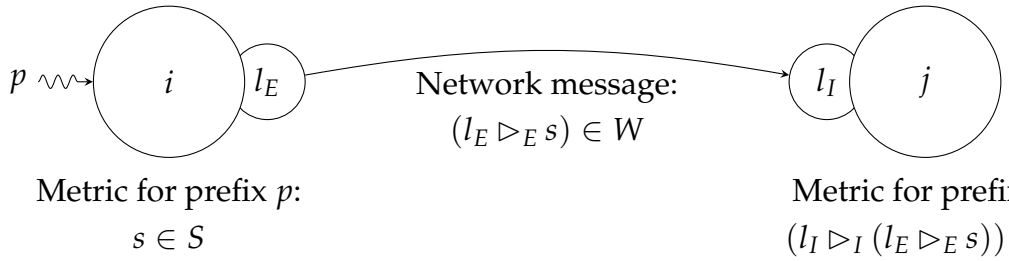
Support for export policy is important for:

- Information hiding: The routes stored by a region's boundary router may contain details of its internal network (e.g. internal paths) which can be commercially sensitive and should not be seen by untrusted external neighbours. These internal details should be erased before exporting to the link, so that there is no chance of leaking them to neighbours. For example, BGP does not transmit its `LOCAL_PREF` attribute (which typically indicates the AS's commercial relationship with adjacent ASes) over EBGP.
- Administrative control: Some parts of policy, e.g. BGP's MED attribute, are designed to be configured locally on the exporting router, to provide a particular limited control over the route selection process to the administrator on the export side – it would defeat the point if the importing router configured these values instead.
- Performance: When routes are filtered out, it is more efficient for the exporting router to filter them, to avoid the CPU and bandwidth costs of needlessly advertising the routes to its neighbour.

Support for import policy is important for:

- Security and robustness: The importing router may not trust its neighbour to only export routes that are permitted under the rules of their commercial relationship. Import filter policies allow the router to control what routes it accepts and readvertises to the wider network, e.g. restricting it to certain destination address prefixes. This is also important for limiting the impact of misconfigurations that would otherwise cause global routing instability [BFMR10].
- Administrative control: Some parts of policy, e.g. BGP's local preference, are designed to be configured locally on the importing router, to provide complete control over the route selection process to the administrator on the import side – it would defeat the point if the exporting router configured these values instead.
- Performance: When applying policy that increases the size of a metric's wire representation (e.g. appending to a path list), applying that policy on import instead of export will reduce the amount of data sent over the network link.

These issues are a critical part of the overall protocol design and are strongly related to the semantics of the routing language; they are not merely implementation details



TYPES

$$\begin{aligned} S &= \text{type of metrics} \\ L &= \text{type of labels} \\ L_E &= \text{type of export labels} \\ L_I &= \text{type of import labels} \\ W &= \text{type of messages on the wire} \end{aligned}$$

FUNCTIONS

$$\begin{aligned} \rhd &\in L \times S \to S && \text{(policy application)} \\ \rhd_E &\in L_E \times S \to W && \text{(export policy application)} \\ \rhd_I &\in L_I \times W \to S && \text{(import policy application)} \\ \odot &\in L_I \times L_E \to L && \text{(policy reconstruction)} \end{aligned}$$

CONSISTENCY

$$(l_I \odot l_E) \rhd s = l_I \rhd_I (l_E \rhd_E s)$$

Figure 5.2: The components of a configuration algebra.

of the routing algorithm. We therefore consider it important to add this export/import interface policy to our model of routing languages.

When extending the metarouting system in this way, it is also important not to sacrifice the benefits that metarouting provides. In particular we want the ability to reason algebraically about the routing language and routing algorithm, and confidence that our implementations correctly match the theoretical model.

5.1.2 Algebraic definition

This section defines the algebraic model of what we will call a *configuration algebra*: an extension of the earlier definition of routing algebra to support the separation of export and import policies, while preserving a bridge between the two algebra models that allows us to re-use the existing metarouting work to construct languages with guaranteed convergence properties.

Our original model used routing algebras with the four components

$$(S,\ L,\ \preceq,\ \triangleright).$$

Figure 5.2 illustrates the components of our configuration algebras. We retain S as the metric type – this is the value stored in routing tables inside each router and is unaffected by our new model. We also make no changes to $\preceq$ – route preferences remain the same. We split the arc label type L into two separate types: the *export label* L_E and the *import label* L_I. Similarly we split the policy application function $\triangleright$ into *export policy application* $\triangleright_E$ and *import policy application* $\triangleright_I$. The type W corresponds to the data type that is shared over the wire by the vectoring protocol; this may differ from S.

To preserve the bridge between the new configuration algebra and our old routing algebra model, we include the original L and $\triangleright$ as part of the configuration algebra. We add an operation $\odot$ that can reconstruct an arc label from adjacent interface configurations, and we insist on *consistency*: the equation

$$(l_I \odot l_E) \triangleright s = l_I \triangleright_I (l_E \triangleright_E s)$$

must be valid for all label and metric values. In other words, we can reconstruct a label $l = l_I \odot l_E$ from bits of configuration data at each end of a link, and the metric

$$s' = l \triangleright s$$

will always be the same as first computing

$$w = l_E \triangleright_E s$$

and then computing

$$s' = l_I \triangleright_I w.$$

We thus have a configuration algebra of the form

$$((S,\ L,\ \preceq,\ \triangleright),\ (L_E,\ L_I,\ W,\ \triangleright_E,\ \triangleright_I,\ \odot))$$

and can represent a network as a graph (V, E) and a weight function $w : E \to (L_E \times L_I)$ (each edge has an export/import pair of labels). We can implement a routing algorithm based on generalised distributed Bellman-Ford which computes $l_I \triangleright_I (l_E \triangleright_E s)$ when a metric is advertised over a link, with half the computation being performed by one router and half by the other. Because of the consistency constraint, we can map this network and algorithm onto a network with a weight function

$$w'(e) = l_I \odot l_E \ \text{ where } \ (l_E, l_I) = w(e)$$

and an algorithm which performs the traditional DBF computation $l \triangleright s$, with the guarantee that these two methods will produce the same result after every step. Any

properties proved of convergence behaviour with the conceptual arc configuration will therefore apply equally to the implementation of the interface configuration.

Although it is trivially possible to construct a consistent configuration algebra for any given $L_E, L_I, \triangleright_E, \triangleright_I$ (just choose $L = L_I \times L_E$ and define $\odot$ and $\triangleright$ such that $l_I \odot l_E = (l_I, l_E)$ and $(l_I, l_E) \triangleright s = l_I \triangleright_I (l_E \triangleright_E s)$), the challenge is to construct ones for which we still can automatically infer the algebraic properties of $(S,\ L,\ \preceq,\ \triangleright)$.

For example, consider the routing algebra that is a direct product of two shortest-paths algebras

$$(\mathbb{N}^+,\ \mathbb{N}^+,\ \leq,\ +) \times (\mathbb{N}^+,\ \mathbb{N}^+,\ \leq,\ +).$$

Each component algebra is *increasing* ($s < l + s$), so the direct product is also increasing. Now imagine we want to specify the first component on export and the second on import, but perform both additions on import:

$$\begin{aligned} S &= \mathbb{N}^+ \times \mathbb{N}^+ \\ L_E &= \mathbb{N}^+ \\ L_I &= \mathbb{N}^+ \\ W &= \mathbb{N}^+ \times \mathbb{N}^+ \times \mathbb{N}^+ \\ l_E \triangleright_E (s_a,\ s_b) &= (s_a,\ s_b,\ l_E) \\ l_I \triangleright_I (w_a,\ w_b,\ l_E) &= (l_E + w_a,\ l_I + w_b). \end{aligned}$$

We cannot say that $\triangleright_E$ or $\triangleright_I$ are increasing, since the order $\preceq$ cannot be applied to values of type W. Therefore we cannot use their properties to determine that their composition $l_I \triangleright_I (l_E \triangleright_E s)$ *is* increasing. We could prove this property by hand or perhaps with an automated theorem prover, but these methods are slow and may not find an answer. To avoid this problem of composing $\triangleright$ from $\triangleright_I$ and $\triangleright_E$ and having to determine its properties afterwards, we will instead construct all of the configuration algebra's components in parallel so we can use our standard property inference rules to understand the final result.

5.1.3 Base configurations

To build provably-correct consistent configuration algebras, we will extend RAML's approach of applying a pre-defined set of constructors to simpler routing language expressions, constrained only by algebraic properties. Given a routing algebra $A = (S,\ L,\ \preceq,\ \triangleright)$ we can define functions to generate the basic configuration algebras, listed in Figure 5.3.

For example, if $A = (\mathbb{N}^+, \mathbb{N}^+, \leq, +)$ then $C_{\text{import\_only}}(A) = (\mathbf{1}, \mathbb{N}^+, \mathbb{N}^+, \text{right}, +, \text{left})$. With this constructor, L_E is the unit type and $\triangleright_E$ is effectively the identity function over S; all the work is shifted to the import interface. Conversely, $C_{\text{export\_only}}$ shifts L and $\triangleright$ to the export interface, with the identity function on import.

C	$=$	(	L_E,	L_I,	W,	$\rhd_E$,	$\rhd_I$,	$\odot$	)
$C_{\text{import\_only}}(A)$	$=$	(	$\mathbf{1}$,	L,	S,	right,	$\rhd$,	left	)
$C_{\text{export\_only}}(A)$	$=$	(	L,	$\mathbf{1}$,	S,	$\rhd$,	right,	right	)
$C_{\text{export\_import}}(A)$	$=$	(	S,	S,	S,	$\rhd$,	$\rhd$,	$\rhd$	)
$C_{\text{left}}(A)$	$=$	(	$\mathbf{1}$,	S,	$\mathbf{1}$,	left,	left,	left	)
$C_{\text{right}}(A)$	$=$	(	$\mathbf{1}$,	$\mathbf{1}$,	S,	right,	right,	right	)

Figure 5.3: Basic configuration algebra constructors.

$C_{\text{export\_import}}$ copies L and $\rhd$ to *both* interfaces: with the shortest paths algebra this means both sides of the link can specify a value to add to the metric, equivalent to a single addition of the sum of those values. This requires $L = S$ for the types to work correctly.

C_{left} replaces the metric with a value that is configured on the import interface. The algebra $C_{\text{left}}(S,\ L,\ \preceq,\ \rhd)$ is equivalent to $C_{\text{import\_only}}(S,\ S,\ \preceq,\ \text{left})$ in its configuration and its result, and may seem redundant, but it differs in one detail: the type of data sent over the wire, W, becomes the unit type $\mathbf{1}$. This makes it valuable in cases where the old metric value should be hidden from the importing router, as with BGP's `LOCAL_PREF`, as well as being a minor performance improvement.

C_{right} has no labels and passes the metric through unchanged. $C_{\text{right}}(S,\ L,\ \preceq,\ \rhd)$ is equivalent to $C_{\text{import\_only}}(S,\ \mathbf{1},\ \preceq,\ \text{right})$ and *is* redundant, but is included here for symmetry.

This set of basic configuration algebras is not necessarily complete but it covers the most important uses discussed in Section 5.1.1; other configuration algebras may be useful in certain situations but will follow the same patterns.

To safely use the C returned by these constructors in a configuration language $(A,\ C)$, we want to ensure the consistency property is satisfied.

To see that $(A,\ C)$ with $A = (S,\ L,\ \preceq,\ \rhd)$ and $C = C_{\text{import\_only}}(A)$ is consistent, we can use the mapping $l_I = l$ and $l_E = \bot$. This mapping of interface labels to arc labels is a bijection, but in general any function $(L_I, L_E) \to L$ will suffice (there may be some arc labels that cannot be implemented with interface labels). Then

$$\begin{aligned}(l_I \odot l_E) \rhd s &= (l \text{ left } \bot) \rhd s \\ &= l \rhd s \\ &= l \rhd (\bot \text{ right } s) \\ &= l_I \rhd_I (l_E \rhd_E s)\end{aligned}$$

so the consistency equation holds.

We can similarly show that

$$C_{\text{export\_only}}(A) \text{ is consistent when } A = (S,\ L,\ \preceq,\ \rhd)$$
$$C_{\text{left}}(A) \text{ is consistent when } A = (S,\ S,\ \preceq,\ \text{left})$$
$$C_{\text{right}}(A) \text{ is consistent when } A = (S,\ \mathbf{1},\ \preceq,\ \text{right}).$$

Further, we can show that $C_{\text{export\_import}}(A)$ is consistent when $A = (S,\ S,\ \preceq,\ \rhd)$ if and only if $\rhd$ is associative ($s_1 \rhd (s_2 \rhd s_3) = (s_1 \rhd s_2) \rhd s_3$ for all $s_1, s_2, s_3 \in S$). In this case the label mapping is not a bijection: for example if $A = (\mathbb{N}^+,\ \mathbb{N}^+,\ \leq,\ +)$, the lowest interface labels give $l = l_I \odot l_E = 1 + 1 = 2$, and it is impossible to implement the arc label $l = 1$.

Note that in all cases except for $C_{\text{import\_only}}$ and $C_{\text{export\_only}}$, the consistency is with a restricted subset of routing algebras. To implement this in RAML with automatic correctness guarantees, we will need to either design any RAML extensions carefully to make the algebra have the necessary form by construction, or else verify the form and report an error if it is incorrect. Since the restrictions in these constructors are either requiring equality between sets or operators, or requiring algebraic properties (namely associativity) that RAML already computes, this verification is straightforward.

5.1.4 Configuration constructors

The base configurations introduced in the previous section could be applied to a complex routing algebra (such as a BGP-style lexicographic product of attributes) as a whole, but this provides very little flexibility. To better match the policy configuration style of BGP, it is important to be able to choose import/export configuration independently for each attribute before combining them into the product. We will therefore take our existing constructors over routing algebras, and extend them to work over configuration algebras.

First we define the product of the C component of a configuration algebra $(A,\ C)$:

$$\begin{aligned} & (L_{E1},\ L_{I1},\ W_1,\ \rhd_{E1},\ \rhd_{I1},\ \odot_1) \\ \times\ & (L_{E2},\ L_{I2},\ W_2,\ \rhd_{E2},\ \rhd_{I2},\ \odot_2) \\ =\ & (L_{E1} \times L_{E2},\ L_{I1} \times L_{I2},\ W_1 \times W_2,\ \rhd_E,\ \rhd_I,\ \odot) \end{aligned}$$

$$\begin{aligned} (l_{E1},\ l_{E2}) \rhd_E (s_1,\ s_2) &= (l_{E1} \rhd_{E1} s_1,\ l_{E2} \rhd_{E2} s_2) \\ (l_{I1},\ l_{I2}) \rhd_I (w_1,\ w_2) &= (l_{I1} \rhd_{I1} w_1,\ l_{I2} \rhd_{I2} w_2) \\ (l_{I1},\ l_{I2}) \odot (l_{E1},\ l_{E2}) &= (l_{I1} \odot_1 l_{E1},\ l_{I2} \odot_2 l_{E2}) \end{aligned}$$

Now we can define direct product and lexicographic product of configuration algebras, which differ only in the $\preceq$ component of their routing algebra:

$$\begin{aligned}(A_1,\ C_1) \times (A_2,\ C_2) &= (A_1 \times A_2,\ C_1 \times C_2)\\ (A_1,\ C_1) \vec{\times} (A_2,\ C_2) &= (A_1 \vec{\times} A_2,\ C_1 \times C_2)\end{aligned}$$

With these product constructors, the configuration semantics of each component is preserved. Given our example of $A = (\mathbb{N}^+,\ \mathbb{N}^+,\ \leq,\ +)$, the configuration algebra

$$(A,\ C_{\text{export\_only}}(A)) \times (A,\ C_{\text{import\_only}}(A))$$

will have $L_E = \mathbb{N}^+ \times \mathbf{1}$ configuring the first component on export, and $L_I = \mathbf{1} \times \mathbb{N}^+$ configuring the second on import.

The function union constructor is slightly more complex. Recall that this combines two algebras with the same S and $\preceq$, merging their distinct label types (and their corresponding $\rhd$) in a disjoint union, for example supporting links tagged as either external or internal in a region-based algebra. We will define this as

$$(A_1,\ C_1) \uplus (A_2,\ C_2) = (A_1 \uplus A_2,\ C_1 \uplus C_2)$$

where $\uplus$ applied to routing algebras is as defined earlier, and $\uplus$ applied to the configuration component is

$$\begin{aligned}&(L_{E1},\ L_{I1},\ W_1,\ \rhd_{E1},\ \rhd_{I1},\ \odot_1)\\ \uplus\ &(L_{E2},\ L_{I2},\ W_2,\ \rhd_{E2},\ \rhd_{I2},\ \odot_2)\\ =\ &(L_{E1} \uplus L_{E2},\ L_{I1} \uplus L_{I2},\ W_1 \uplus W_2,\ \rhd_E,\ \rhd_I,\ \odot)\end{aligned}$$

$$\begin{aligned}\text{inl}(l_{E1}) \rhd_E s &= \text{inl}(l_{E1} \rhd_{E1} s)\\ \text{inr}(l_{E2}) \rhd_E s &= \text{inr}(l_{E2} \rhd_{E2} s)\\ \text{inl}(l_{I1}) \rhd_I \text{inl}(w_1) &= l_{I1} \rhd_{I1} w_1\\ \text{inr}(l_{I2}) \rhd_I \text{inr}(w_2) &= l_{I2} \rhd_{I2} w_2\\ \text{inl}(l_{I1}) \rhd_I \text{inr}(w_2) &= \text{error}\\ \text{inr}(l_{I2}) \rhd_I \text{inl}(w_1) &= \text{error}\\ \text{inl}(l_{I1}) \odot \text{inl}(l_{E1}) &= \text{inl}(l_{I1} \odot_1 l_{E1})\\ \text{inr}(l_{I2}) \odot \text{inr}(l_{E2}) &= \text{inr}(l_{I2} \odot_2 l_{E2})\\ \text{inl}(l_{I1}) \odot \text{inr}(l_{E2}) &= \text{error}\\ \text{inr}(l_{I2}) \odot \text{inl}(l_{E1}) &= \text{error}\end{aligned}$$

Labels L_E, L_I and wire metrics W are each tagged with either inl or inr (injected into left or right side of the disjoint union type). $\rhd_E$ produces a wire metric with the same tag as the export label. $\rhd_I$ then requires the wire metric to have the same tag as the import label; if not (e.g. one router has tagged a link as 'external' while another has tagged the same link as 'internal') then it is impossible to perform the computation of $\rhd_I$, so this is an error. This possibility of inconsistency is an inevitable consequence of spreading data across a distributed system; we discuss a number of ways to prevent or to handle the error case later in this chapter.

We will also need to extend **addtop** to apply to configuration algebras. This adds a new constant value c that is least preferred in the order, and acts as an annihilator ($c \rhd s = l \rhd c = c$): a link labelled with c will cause the route's metric to become and remain c forever, effectively filtering out the route. Filters on export and import can both be useful; we could extend **addtop** to configuration algebras by defining variants that allow filters on just one or the other, but this would be needless complexity, so instead we extend the constructor to allow either (or both) of the interface labels to be c and filter out the route. The functions $\rhd_E$, $\rhd_I$ and $\odot$ are all extended to simply propagate c.

$$\mathbf{addtop}(c,\ (L_E,\ L_I,\ W,\ \rhd_E,\ \rhd_I,\ \odot)) \quad = \quad (L_E \cup \{c\},\ L_I \cup \{c\},\ W_1 \cup \{c\},\ \rhd'_E,\ \rhd'_I,\ \odot')$$

$$\begin{aligned}
c \rhd'_E s &= c \\
l_E \rhd'_E c &= c \\
l_E \rhd'_E s &= l_E \rhd_E s \\
c \rhd'_I w &= c \\
l_I \rhd'_I c &= c \\
l_I \rhd'_I w &= l_I \rhd_I w \\
c \odot' l_E &= c \\
l_I \odot' c &= c \\
l_I \odot' l_E &= l_I \odot l_E
\end{aligned}$$

All of these constructors must preserve the consistency condition, so that we can ensure the consistency of any configuration algebra specification constructed from the consistent base configurations. This is straightforward as long as the error case in the function union is avoided.

5.2 Extending RAML

There are several different ways we could extend RAML to construct configuration languages using the concepts defined above. This section will detail two models: a tightly coupled approach where the configuration is integrated with the routing language specification, and a loosely coupled approach where the configuration is determined by external annotations. The difference in these two approaches is purely one of metalanguage design – both will be translated into the same algebraic model of configuration languages, but it is useful to contrast the design choices that can be made.

raml-base ::= `int_min_plus(`*erl-ty-int-unbounded*`)`
| `int_range_min_plus(`*num*`, `*num*`)`
| `int_max_min(`*erl-ty-int*`)`
| `paths(`*erl-ty*`)`
| `strata(`*num*`, `*num*`, `*num...*`)`

ramlc-exp ::= *raml-base*
| `add_top(`*name*`, `*ramlc-exp*`)`
| `lex_product(`*name*`=`*ramlc-exp*`, `*name*`=`*ramlc-exp*`, ...)`
| `function_union(`*name*`=`*ramlc-exp*`, `*name*`=`*ramlc-exp*`, ...)`
| `right(`*ramlc-exp*`)`
| `left(`*ramlc-exp*`)`
| `export(`*ramlc-exp*`)`
| `export_import(`*ramlc-exp*`)`

Figure 5.4: Grammar for RAML$^{\mathcal{C}}$.

5.2.1 Tightly-coupled syntax

We will define a new version of RAML and name it RAML$^{\mathcal{C}}$, with extended syntax to define configuration algebras. The grammar for the metalanguage is defined in Figure 5.4. Compare this to the syntax in Section 3.2 – we have kept the same *raml-base* but have extended *raml-exp* to include `export` and `export_import` keywords.

We need to replace the old semantic function $[\![$*raml-exp*$]\!]$, which gave a routing algebra, with a new semantic function $[\![$*ramlc-exp*$]\!]^{\mathcal{C}}$ giving a configuration algebra. An expression e will now map to $[\![e]\!]^{\mathcal{C}} = ((A,\ P),\ C)$, where $A = (S,\ L,\ \preceq,\ \triangleright)$ is the routing algebra as before, P is the set of algebraic properties for A, and C is a tuple $(L_E,\ L_I,\ W,\ \triangleright_E,\ \triangleright_I,\ \odot)$ capturing the other components of a configuration language as described in Figure 5.2. We want to guarantee consistency by construction for any expression in *ramlc-exp*.

We define the algebraic semantics in Figure 5.5. (Algebraic property inference rules are the same as with the original RAML, and not repeated here.)

These constructors can be grouped into four categories. First, any *raml-base* is given import-only configuration semantics. This is reasonable default behaviour, and it allows any RAML specification to be interpreted as RAML$^{\mathcal{C}}$ with no changes.

Second, `left` applies the **left** routing algebra constructor at the same time as applying C_{left} for the configuration. The consistency condition for C_{left} requires the algebra to be of the form produced by **left**, so this is guaranteed by performing both operations together. `right` acts similarly.

Third, `lex_product` and `function_union` combine the configuration components C_1, C_2

$$[\![\textit{raml-base}]\!]^{\mathcal{C}} = ((A,\ P),\ C_{\text{import\_only}}(A)) \quad \text{where } (A,\ P) = [\![\textit{raml-base}]\!]$$

$$[\![\texttt{add\_top}(n,\ e)]\!]^{\mathcal{C}} = ((\mathbf{addtop}(n,\ A),\ \{\ldots\}),\ \mathbf{addtop}(n,\ C)) \quad \text{where } ((A,\ P),\ C) = [\![e]\!]^{\mathcal{C}}$$

$$[\![\texttt{lex\_product}(n_1\texttt{=}e_1,\ n_2\texttt{=}e_2)]\!]^{\mathcal{C}} = ((A_1 \mathbin{\vec{\times}} A_2,\ \{\ldots\}),\ C_1 \times C_2) \quad \text{where } ((A_1,\ P_1),\ C_1) = [\![e_1]\!]^{\mathcal{C}},\ ((A_2,\ P_2),\ C_2) = [\![e_2]\!]^{\mathcal{C}}$$

$$[\![\texttt{function\_union}(n_1\texttt{=}e_1,\ n_2\texttt{=}e_2)]\!]^{\mathcal{C}} = ((A_1 \uplus A_2,\ \{\ldots\}),\ C_1 \uplus C_2) \quad \text{where } ((A_1,\ P_1),\ C_1) = [\![e_1]\!]^{\mathcal{C}},\ ((A_2,\ P_2),\ C_2) = [\![e_2]\!]^{\mathcal{C}}$$

$$[\![\texttt{right}(e)]\!]^{\mathcal{C}} = ((\mathbf{right}(A),\ \{\ldots\}),\ C_{\text{right}}(\mathbf{right}(A))) \quad \text{where } ((A,\ P),\ C) = [\![e]\!]^{\mathcal{C}}$$

$$[\![\texttt{left}(e)]\!]^{\mathcal{C}} = ((\mathbf{left}(A),\ \{\ldots\}),\ C_{\text{left}}(\mathbf{left}(A))) \quad \text{where } ((A,\ P),\ C) = [\![e]\!]^{\mathcal{C}}$$

$$[\![\texttt{export}(e)]\!]^{\mathcal{C}} = ((A,\ P),\ C_{\text{export\_only}}(A)) \quad \text{where } ((A,\ P),\ C) = [\![e]\!]^{\mathcal{C}}$$

$$[\![\texttt{export\_import}(e)]\!]^{\mathcal{C}} = ((A,\ P),\ C_{\text{export\_import}}(A)) \quad \text{where } ((A,\ P),\ C) = [\![e]\!]^{\mathcal{C}}$$

Figure 5.5: Algebraic semantics for RAML$^{\mathcal{C}}$.

of the sub-algebras as defined in Section 5.1.4. Similarly, `add_top` adds the constant value to its sub-expression's algebra. These are the only expressions that use the Cs to build more complex configuration algebras, instead of discarding them and defining a new C based only on the A components.

Finally, `export` and `export_import` replace only the C component and leave A unchanged; these are taking a routing algebra and moving its configuration to different sides of the network link without affecting the semantics of the routing algebra. While `export` can be used on any arbitrary routing algebra A, `export_import` will require the RAML compiler to verify that the algebra has $L = S$ and that it is associative.

We must also define the ERL semantics $(\!|\textit{ramlc-exp}|\!)^{\mathcal{C}}$ of a configuration language expressed in RAML. This is generally a straightforward mapping, mirroring the alge-

braic semantics – for example we can define $(\!|\texttt{export}(e)|\!)^{\mathcal{C}}$ as

```
sig = (|e|)^C_sig
lbl = (|e|)^C_lbl
ord = (|e|)^C_ord
tfm = (|e|)^C_tfm
wir = (|e|)^C_sig
lbl_e = (|e|)^C_lbl
lbl_i = TyUnit
tfm_e = (|e|)^C_tfm
tfm_i = fun (li, s) -> s
rec = fun (li, le) -> le
```

where `sig`, `lbl`, `ord` and `tfm` have the same meanings as with the original RAML definition in Section 3.2. The new values are the wire metric `wir` (W), the policy reconstruction function `rec` ($\odot$), and the import/export variants of label and policy application, which are all implementations of the corresponding component of the algebraic semantics. A typical distributed vector routing algorithm will never use the code generated for `lbl`, `tfm` or `rec`, but we include them to support the recursive definition of semantics.

We now return to the scoped product example of Section 3.3.3. We want to split the computation associated with external links in a BGP-like manner: the `ecomm` and `epath` attributes should be computed on export, while the `idist` and `ipath` should not be transmitted outside the region and should be replaced by new values on import. On internal links, `ipath` is computed on import, but both import and export interfaces can add a cost to `idist`. We accomplish this in Figure 5.6 by using the new operations of the language (in bold).

We can apply the algebraic semantics defined earlier to extract the behaviour of the two policy application functions, $\triangleright_E$ and $\triangleright_I$. First we handle the annihilator value `W` (expressed as ∞):

$$\begin{aligned} \infty \triangleright_E s &= \infty \\ l_E \triangleright_E \infty &= \infty \\ \infty \triangleright_I w &= \infty \\ l_I \triangleright_I \infty &= \infty \end{aligned}$$

Next we have the case where the labels represent an internal link:

$$\begin{aligned} \text{inl}(\bot,\ \bot,\ v,\ \bot) \ \triangleright_E\ (ec,\ ep,\ id,\ ip) &= \text{inl}(ec,\ ep,\ v + id,\ ip) \\ \text{inl}(\bot,\ \bot,\ v,\ (i,\ j)) \ \triangleright_I\ \text{inl}(ec,\ ep,\ id,\ ip) &= (ec,\ ep,\ v + id,\ (i,\ j) \triangleright_{paths} ip) \end{aligned}$$

Next we have labels for an external link:

$$\begin{aligned} \text{inr}(x,\ (m,\ n),\ \bot,\ \bot) \ \triangleright_E\ (ec,\ ep,\ id,\ ip) &= \text{inr}(x \triangleright_{cpp} ec,\ (m,\ n) \triangleright_{paths} ep,\ \bot,\ \bot) \\ \text{inr}(\bot,\ \bot,\ v,\ l) \ \triangleright_I\ \text{inr}(ec,\ ep,\ \bot,\ \bot) &= (ec,\ ep,\ v,\ l) \end{aligned}$$

```
add_top(W,
  function_union(
    internal =
      lex_product(
        ecomm = right(cpp),
        epath = right(paths(TyIntNonNeg)),
        idist = export_import(int_min_plus(TyIntNonNeg)),
        ipath = paths(TyIntNonNeg)
      ),
    external =
      lex_product(
        ecomm = export(cpp),
        epath = export(paths(TyIntNonNeg)),
        idist = left(int_min_plus(TyIntNonNeg)),
        ipath = left(paths(TyIntNonNeg))
      )
  )
```

Figure 5.6: Scoped product specification with configuration constructors.

Finally we have the case where the two sides of the link disagree on whether it is internal or external; our semantics for `function_union` considers this to be an error. One option would be for the routing algorithm implementation to detect the runtime error and report an error message to the administrator and tear down the incorrectly-configured connection over that link. A more robust option would be for the implementation to verify consistency when setting up the connection between routers, and refuse to connect unless the configuration is correct, so the error cases can never be encountered at runtime. Another alternative would be to return ∞ for errors:

$$\begin{aligned} \text{inl}(\bot,\ \bot,\ v,\ (i,\ j)) \ \triangleright_I\ \text{inr}(ec,\ ep,\ \bot,\ \bot) &= \infty \\ \text{inr}(\bot,\ \bot,\ v,\ l) \ \triangleright_I\ \text{inl}(ec,\ ep,\ id,\ ip) &= \infty \end{aligned}$$

This could be achieved by special-casing the semantics of the RAML expression pattern "`add_top(`c`, function_union(...))`" to return c in error cases.

5.2.2 Loosely-coupled syntax

The definition in the previous section is a direct extension of the RAML syntax from routing algebras to configuration algebras. However, it introduces a number of practical problems due to conflating the specification of the routing algebra and its configuration behaviour.

When we compile a specification, the compiler must compute and verify the algebraic properties of the routing algebra. When it finds errors, it will report them in terms of the routing algebra (e.g. "this specific example of a label L violates the increasing property"), not the configuration algebra ("this combination of L_E and L_I violates the property"); in general it is not possible to map from arc labels to interface labels. To understand the errors, users will therefore need a mental model corresponding to the routing algebra – we cannot hide that concept from them, but the syntax fails to make it explicit, potentially increasing confusion.

Another issue is that routing algebra correctness relies on it being used identically across a whole network, whereas configuration algebra correctness depends only on agreement between a pair of consenting peers – the splitting of labels across export and import interfaces can be different for every link. A protocol designer will have to write down a specific routing algebra and tell every router to use it, but if that specification is inextricably linked with configuration details, the protocol designer will have been forced to decide those configuration details and take power away from each pair of peers. Instead, the designer should just specify the routing algebra by itself, and peers should independently specify configuration algebras that are compatible with the routing algebra. Determining "compatible" based on the compiled form of a specification is hard – instead, as with proving correctness, we should guarantee compatibility by construction, so that we can produce multiple configuration algebras that are known to implement the same routing algebra.

One way to do this is to define the configuration algebra as a RAML (routing algebra) specification plus a separate list of annotations. We will need to define the semantics of this so that any valid set of annotations combined with a routing algebra will give a compatible configuration algebra.

5.2.3 Loosely-coupled syntax specification

We can use a simple syntax to express external annotations over a RAML specification. The RAML syntax defines a tree structure, and every node in the syntax tree with two or more child expressions has *labelled* branches: these are the `lex_product` and `function_union` terms. Other terms such as `add_top` and `left` have a single unlabelled child expression.

For configuration, we don't need to be able to annotate every node in the syntax. As with the semantics for RAML$^{\mathcal{C}}$, `left` and `right` will ignore the configuration component of their child expression and replace it with $C_{\text{left}}(A)$ or $C_{\text{right}}(A)$, so there is no value in giving configuration annotations for the child expressions. However, `add_top` extends its child's configuration; given a configuration algebra $((A,\ P),\ C)$ there is a useful distinction between $C_{\text{export\_only}}(\mathbf{addtop}(n,\ C))$ (allowing the filtering constant n

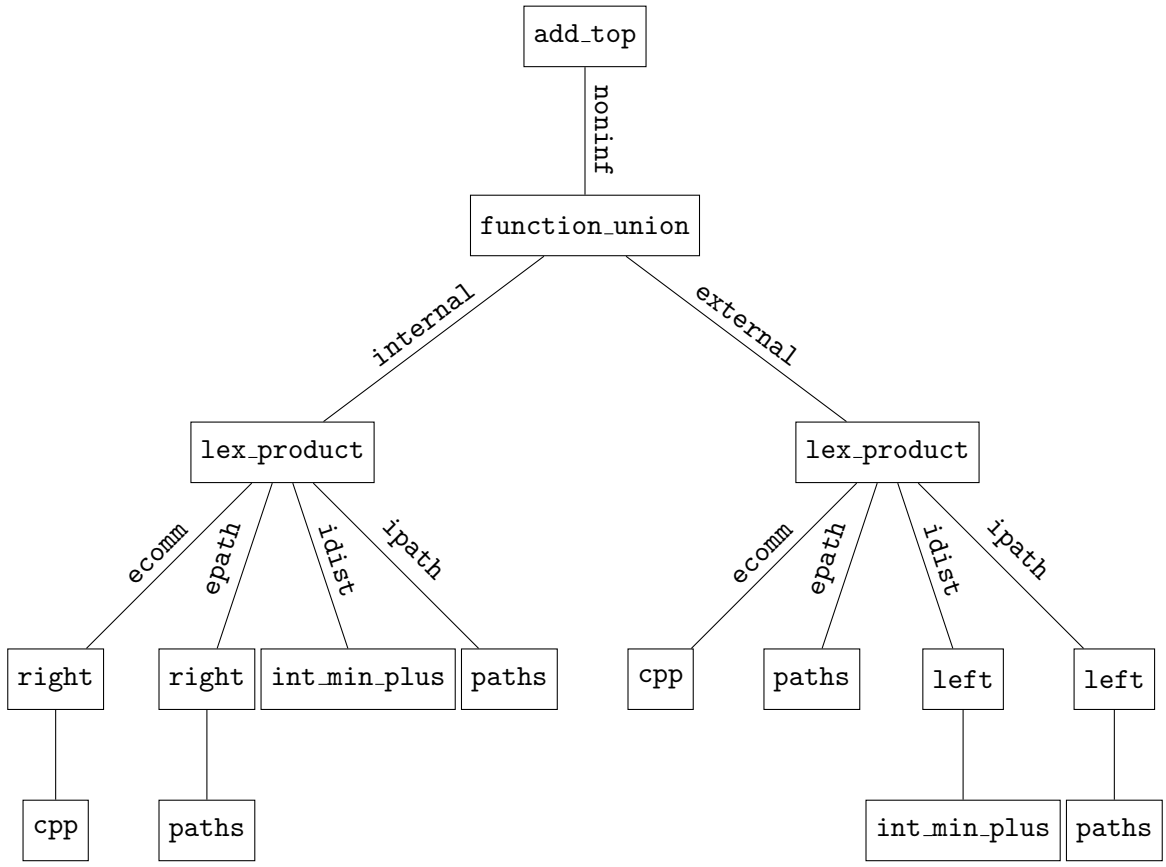


Figure 5.7: Annotatable syntax tree for scoped product example.

only on export) and **addtop**$(n,\ C_{\text{export\_only}}(A))$ (allowing n on both interfaces). We will therefore add an artificial label "noninf" for the child expression of `add_top`, allowing it to be addressed directly.

Now any node that may need annotations can be addressed by a sequence of labels. The annotation syntax then consists of associating a configuration keyword (*export*, *import*, or *export_import*) with a label sequence.

Figure 5.7 illustrates the syntax tree with labelled nodes for our scoped product example. We will express annotations in the following form:

```
export_import(noninf.internal.idist)
export(noninf.external.ecomm)
export(noninf.external.epath)
```

We want to use these annotations in our definition of the semantics of an annotated RAML specification. We therefore start by converting the annotation syntax into a form that will simplify our definition. In this form, an annotation is either a configuration keyword or a named map of annotations. We represent a named map as a set of $(name, value)$ tuples, where the name is required to be unique within its set.

The interpretation of the example above is:

{
  (“noninf”, {
    (“internal”, {
      (“idist”, *export_import*)
    }),
    (“external”, {
      (“ecomm”, *export*),
      (“epath”, *export*)
    })
  })
}

Parsing into this form is straightforward; we do not define it here.

We use exactly the same RAML specification language as defined in Section 3.2. We make use of exactly the same $[\![\textit{raml-exp}]\!]$ semantics over RAML syntax. However, we define a new augmented semantics $[\![\textit{raml-exp}; \textit{annot}]\!]^{\mathcal{A}}$ over RAML syntax plus configuration annotations, which gives either an error or a configuration algebra.

We first define semantics for arbitrary RAML syntax combined with the standard annotation keywords:

$$\begin{aligned}
[\![e; \textit{export}]\!]^{\mathcal{A}} &= ((A,\ P),\ C_{\text{export}}(A)) \\
[\![e; \textit{import}]\!]^{\mathcal{A}} &= ((A,\ P),\ C_{\text{import}}(A)) \\
[\![e; \textit{export_import}]\!]^{\mathcal{A}} &= ((A,\ P),\ C_{\text{export_import}}(A)) \\
&\quad \text{where } (A,\ P) = [\![e]\!]
\end{aligned}$$

We also define the augmented semantics of the RAML constructors, in Figure 5.8. The nested constructors (`lex_product` and `function_union`) ‘unwrap’ the annotations at the same time as their named arguments. The annotations and RAML syntax both define a tree structure, and we are recursing down both trees in parallel. Note that we use the ‘import’ configuration semantics when only one of the names is present in the annotation. This is implementing a default to simplify the specifications: the annotations need only refer to a subset of the fields of the product or union, and the unannotated branches of the routing language will simply use ‘import’.

For `lex_product`, it is possible that none of the above patterns will match. Our annotations are defined to be either a keyword, or a non-empty set of named annotations. The only case that will not match any patterns is when there is a set with at least one named annotation which does not match either of the names in the `lex_product`. (There may or may not be other correctly-named annotations in the set.) This indicates

$$[\![\texttt{add\_top}(n,\ e);\ \{(\text{“noninf”}, c)\}]\!]^{\mathcal{A}} = ((\mathbf{addtop}(n,\ A),\ \{\ldots\}),\ \mathbf{addtop}(n,\ C))$$
$$\text{where}\quad [\![e;\ c]\!]^{\mathcal{A}} \ = \ ((A,\ P),\ C)$$

$$[\![\texttt{lex\_product}(n_1\texttt{=}e_1,\ n_2\texttt{=}e_2);\ \{(\text{“}n_1\text{”}, c_1), (\text{“}n_2\text{”}, c_2)\}]\!]^{\mathcal{A}} = ((A_1 \vec{\times} A_2,\ \{\ldots\}),\ C_1 \times C_2)$$
$$\begin{aligned}\text{where}\quad [\![e_1;\ c_1]\!]^{\mathcal{A}} &= ((A_1,\ P_1),\ C_1)\\ [\![e_2;\ c_2]\!]^{\mathcal{A}} &= ((A_2,\ P_2),\ C_2)\end{aligned}$$

$$[\![\texttt{lex\_product}(n_1\texttt{=}e_1,\ n_2\texttt{=}e_2);\ \{(\text{“}n_1\text{”}, c_1)\}]\!]^{\mathcal{A}} = ((A_1 \vec{\times} A_2,\ \{\ldots\}),\ C_1 \times C_2)$$
$$\begin{aligned}\text{where}\quad [\![e_1;\ c_1]\!]^{\mathcal{A}} &= ((A_1,\ P_1),\ C_1)\\ [\![e_2;\ \mathit{import}]\!]^{\mathcal{A}} &= ((A_2,\ P_2),\ C_2)\end{aligned}$$

$$[\![\texttt{lex\_product}(n_1\texttt{=}e_1,\ n_2\texttt{=}e_2);\ \{(\text{“}n_2\text{”}, c_2)\}]\!]^{\mathcal{A}} = ((A_1 \vec{\times} A_2,\ \{\ldots\}),\ C_1 \times C_2)$$
$$\begin{aligned}\text{where}\quad [\![e_1;\ \mathit{import}]\!]^{\mathcal{A}} &= ((A_1,\ P_1),\ C_1)\\ [\![e_2;\ c_2]\!]^{\mathcal{A}} &= ((A_2,\ P_2),\ C_2)\end{aligned}$$

Figure 5.8: Augmented semantics of RAML constructors with configuration annotations. `function_union` is defined analogously to `lex_product`.

a clear error in the annotation specification, referring to non-existent properties, so the specification will be rejected and the incorrect name will be reported to the user. Similarly, an `add_top` without either a keyword or a set containing the named annotation “noninf” will be an error due to an incorrect name.

The `left` and `right` constructors do not have any special semantics defined: their configuration annotation must be a keyword, not a set, and it will be applied to the whole of $\mathbf{left}([\![e]\!])$. The tightly-coupled syntax in the previous section allowed configuration modes to be applied to the nested expression e, but these were discarded when applying the **left** constructor – it reconstructed the configuration algebra based solely on its argument's routing algebra and ignored the rest of the argument's configuration algebra. It is therefore no loss that we cannot express annotations on the nested expression in this new syntax – in fact it simplifies the language by disallowing confusingly meaningless specifications.

From this definition of semantics, it is easy to show that for any RAML specification e and configuration annotations c, we get $[\![e;\ c]\!]^{\mathcal{A}} = ([\![e]\!],\ C)$ for some C. This means that if we have two sets of annotations for the same RAML syntax e, they will both produce configuration algebras that are consistent with the same routing algebra $[\![e]\!]$. This satisfies our goal of allowing a single global routing algebra specification to ensure correct operation of a network, with the potential for local choices of configuration between each pair of routers.

5.3 Non-local configuration

So far, we have split the configuration of a link into two halves – the export policy L_E and computation $\triangleright_E$ on one side, the import policy L_I and computation $\triangleright_I$ on the other – and assumed that the policies will be specified by configuration files on whichever router performs the corresponding computation. This is the natural way to configure a decentralised, distributed network, and is the standard behaviour for all common vector routing protocols. However, in some cases it can be valuable to expand this model and allow a router to specify policy that will be computed by a different router.

There is a recent extension to BGP called Outbound Route Filtering (ORF), which is a step in this direction: part of the policy for a link is specified on the importing router, for which the computation is performed the exporting router. This alters the tradeoffs between the goals we listed in Section 5.1.1. In particular the importing router can specify a set of filters on address prefixes, giving it administrative control over what routes it will accept, but they are computed on the exporting router, giving increased performance due to reduced communication overhead.

BGP ORF special-cases certain types of filtering – most aspects of policy are stuck on the same router for configuration and computation, and the ORF approach is not scalable to handling more complex policy due to its low-level nature and awkward integration with policy languages. We believe there may be value in permitting much more flexibility in this area, and our algebraic model provides a powerful way to reason about this at a higher level. This section discusses a modest extension to the metarouting system to support a superset of the ORF feature.

5.3.1 BGP Outbound Route Filtering

There are currently two RFCs related to BGP outbound route filtering: RFC 5291 [CR08] defines a generic mechanism to implement filters as an extension to the BGP protocol, while RFC 5292 [CS08] defines a specific type of filter based on address prefix matching. We can summarise the behaviour of RFC 5291 as follows:

Two BGP neighbours advertise their ORF capabilities as part of the handshake process when first connecting, listing the address families (e.g. IPv4 or IPv6) and ORF types (e.g. address prefix matching) that they support. The intersection of the advertised features determines what will be allowed during this session.

Either BGP speaker may send a ROUTE-REFRESH message (or series of messages) that is extended to contain a list of ORF entry changes (additions and removals) for its neighbour to apply. Each ORF entry specifies an address family and ORF type, a PERMIT/DENY flag, and a type-specific value. Each router stores the set of ORF entries advertised by its neighbour and updates the set in response to these messages. After

performing the standard BGP decision process to determine which routes to advertise, a route is matched against all ORF entries to get a PERMIT or DENY result, which determines whether it will be advertised to the neighbour. (The ORF type determines how to choose the result if the route matches multiple ORFs of that type. If it matches ORFs of different types, DENY takes precedence over PERMIT.)

RFC 5292 defines one specific ORF type for address prefix matching. The ORF entry value contains a prefix (e.g. 192.168.0.0/16 for IPv4 address families), and optional minimum and maximum lengths. A route containing a destination address prefix will match the ORF if it is equally or more specific than the ORF prefix (e.g. 192.168.0.0/16 will match a route to 192.168.1.0/24, but not a route to 192.0.0.0/8 or to 10.0.0.0/24), and if its destination prefix length is between the specified minimum and maximum. The ORF entry value also contains a sequence number which is assumed to be unique; if multiple ORF entries match then the lowest sequence number determines whether to PERMIT or DENY the route.

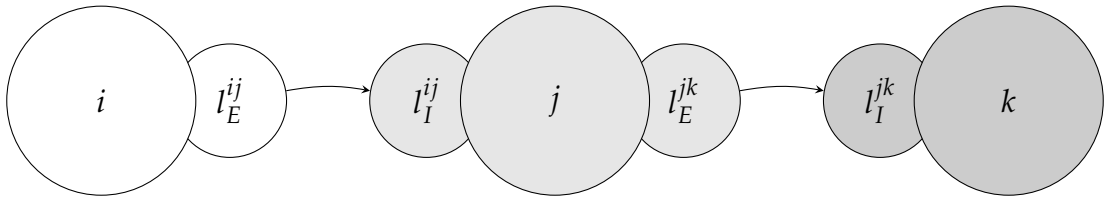
From this, we can see that the ORF mechanism is designed very specifically for filtering: it cannot be used for general policy application without significant design changes, as it is applied after the BGP decision process has completed. Also, the mechanism is very rigid: the address prefix matching works for the cases that the protocol extension designers considered to be common and useful, but there is no way to match against more complex prefix patterns and (more importantly) no way to look at any other route attributes. Given the very low-level nature of ORF, with the functionality being defined in octet layouts and verbose prose, experimenting with new features is a time-consuming process.

Our metarouting approach can be extended to incorporate this functionality and to provide much greater flexibility with much less effort.

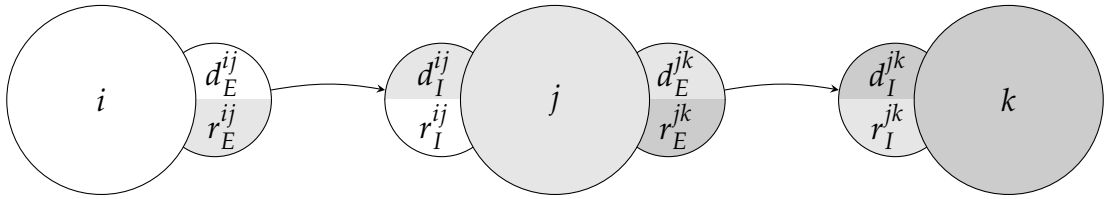
5.3.2 Extending metarouting

For the same reasons as with our original configuration algebras in this chapter, we want an algebraic model that relates back to simpler models that we can already analyse. In this case, the computational aspect of the language is exactly the same as our original languages: the exporting router performs some computation over the route metric and some configuration data, while the importing router does the same. The only difference is that the protocol requires the configuration data used by each computation to come from two sources: part is local, part is remote.

Some configuration data may be used both locally and remotely: for example, an importing router's route filter might be executed remotely on the exporting router to improve performance, but often it should also be executed on the importing router



(a) Local interface configuration.



(b) Shared interface configuration.

Figure 5.9: Associating policy labels with interfaces configured by (a) the node connected directly to that interface, or (b) split into two parts configured by both nodes on the arc. Shading indicates which node controls which part of the configuration data.

to guarantee correct operation even if the exporting router fails to apply the policy correctly (due to software bugs or an intentional attack).

As one of our goals listed earlier was to support information-hiding between neighbours, we should not expect them to share the entirety of their policy with each other. Instead, only the minimal amount necessary for the neighbour's computation should be shared.

We illustrate this model in Figure 5.9. The original configuration algebras are defined purely locally – the labels on each interface are controlled by the associated router. We will extend this so control of the label on an interface is shared by both routers on that link. This shared model can be mapped back onto the local model, providing a path to guarantee correct operation of the protocol, by merging the local and remote parts of the label on each interface into a single label.

We define our new shared configuration algebra as

$$((S,\ L,\ \preceq,\ \rhd),\ (L_E,\ L_I,\ W,\ \rhd_E,\ \rhd_I,\ \odot),\ (D_E,\ D_I,\ R_E,\ R_I,\ \pi_E,\ \pi_I,\ \odot_E,\ \odot_I))$$

which consists of the old local configuration algebra, plus the exporting router's configuration data D_E and importing router's configuration data D_I; the exporting router's remotely-computed configuration data R_E and importing router's R_I; the functions π that project out the remote component R of the configuration data D:

$$\pi_E : D_E \to R_E$$

$$\pi_I : D_I \to R_I$$

and two new label reconstruction functions

$$\odot_E : D_E \times R_I \to L_E$$

$$\odot_I : R_E \times D_I \to L_I$$

with the appropriate consistency constraint.

Now we need to extend our set of constructors to work with this new language, guaranteeing the consistency constraint.

In the simplest case we can set $D_E = L_E$, $D_I = L_I$, $R_E = R_I = \mathbf{1}$, and define $d_E \odot_E r_I = d_E$ and $r_E \odot_I d_I = d_I$. All data will be defined and used locally, as with our local configuration algebra. This will be the default behaviour for a language specification.

We can add some new constructors that make use of remote configuration data:

$C =$	$(\ L_E, L_I, W,\ \rhd_E,\ \rhd_I,\ \odot,\ D_E, D_I, R_E, R_I, \pi_E, \pi_I,\ \odot_E,\ \odot_I\ )$
$C_{\text{import}}(A) =$	$(\ \mathbf{1},\ L,\ S,\ \text{right},\ \rhd,\ \text{left},\ \mathbf{1},\ L,\ \mathbf{1},\ \mathbf{1},\ \mathbf{1},\ \mathbf{1},\ \text{left},\ \text{right}\ )$
$C_{\text{shift\_import}}(A) =$	$(\ L,\ \mathbf{1},\ S,\ \rhd,\ \text{right},\ \text{right},\ \mathbf{1},\ L,\ \mathbf{1},\ L,\ \mathbf{1},\ \text{id},\ \text{right},\ \text{left}\ )$
$C_{\text{dupe\_import}}(A) =$	$(\ L,\ L,\ S,\ \rhd,\ \rhd,\ \text{left},\ \mathbf{1},\ L,\ \mathbf{1},\ L,\ \mathbf{1},\ \text{id},\ \text{right},\ \text{right}\ )$

These three constructors all have the same configuration data D_E, D_I. They also have the same underlying algebra $(S,\ L,\ \preceq,\ \rhd)$. What differs is how the configuration data is split into the labels L_E and L_I, and how those labels are computed with. The first case is the original local application. $C_{\text{shift\_import}}$ leaves the configuration on the importing router, but shifts the application entirely onto the exporting router. $C_{\text{dupe\_import}}$ performs the application $\rhd$ on *both* routers – this imposes the additional requirement that $\rhd$ is idempotent in order to preserve our consistency constraint. These constructors provide functionality similar to BGP ORF, allowing the computation to be shifted without changing the configuration input or the global behaviour.

We can implement this using exactly the same annotation model as before; we simply need to extend the semantics and add the new constructors following the same pattern, and do not go into the details here.

We want to prove that these constructors all implement $(S,\ L,\ \rhd)$ correctly given the same input $(d_i,\ d_e)$, i.e. that they are all equivalent except for the internal communication and computation details.

For any language, we have:

$$\begin{aligned} L \rhd s &= (L_I \odot L_E) \rhd s \\ &= L_I \rhd_I (L_E \rhd_E s) \\ &= ((\pi_E(d_e) \odot_I d_i) \rhd_I ((d_e \odot_E \pi_I(d_i)) \rhd_E s) \end{aligned}$$

For C_{import} we have:

$$\begin{aligned} & ((\pi_E(d_e) \odot_I d_i) \rhd_I ((d_e \odot_E \pi_I(d_i)) \rhd_E s) \\ = \; & ((\mathbf{1} \text{ right } d_i) \rhd ((d_e \text{ left } \mathbf{1}) \text{ right } s) \\ = \; & d_i \rhd s \end{aligned}$$

For $C_{\text{shift\_import}}$ we have:

$$\begin{aligned} & ((\pi_E(d_e) \odot_I d_i) \rhd_I ((d_e \odot_E \pi_I(d_i)) \rhd_E s) \\ = \; & ((\mathbf{1} \text{ left } d_i) \text{ right } ((d_e \text{ right } d_i) \rhd s) \\ = \; & d_i \rhd s \end{aligned}$$

For $C_{\text{dupe\_import}}$ we have:

$$\begin{aligned} & ((\pi_E(d_e) \odot_I d_i) \rhd_I ((d_e \odot_E \pi_I(d_i)) \rhd_E s) \\ = \; & ((\mathbf{1} \text{ right } d_i) \rhd ((d_e \text{ right } d_i) \rhd s) \\ = \; & d_i \rhd (d_i \rhd s) \\ = \; & d_i \rhd s \qquad \text{iff } \rhd \text{ idempotent} \end{aligned}$$

These all produce the same behaviour for the same input d_i and s, so they are all equivalent implementations of the routing algebra and make no difference to how the network will be configured by users, but they can make an important difference to concerns such as performance, security, and information-hiding.

5.4 Remaining issues

This chapter aimed to bridge the gap between routing algebras based on arc labels, and distributed implementations where policy is applied at interfaces. The algebraic models and extensions to the metarouting specification languages show that much of the desired behaviour can be handled in this way. However, given the scope of the problem, there are a number of details that remain unresolved; this section will describe several issues that remain.

5.4.1 Attribute naming

In RAML and ERL, we allow the fields of products and disjoint unions to have human-readable names. This is important for readability of the routing language specifications, and also for use in the syntax for router configurations. In the scoped product example of Section 3.3.3, we use the specified names such as "`epath`" for both metric and label types. However, this is an over-simplified approach that can cause unintuitive behaviour in configuration syntax: the `epath` field of a metric is (as the name

suggests) a path, but the `epath` field of a label is only a single element that will be prepended to the path. It may be valuable to allow a routing language specification to rename this field to a different name (perhaps "`enum`") in the label type independently of the metric type, similarly to how BGP uses the terms "AS number" (in configuration) and "AS path" (in metrics). As the names are ignored entirely by the algebraic semantics of a RAML or ERL expression, this renaming will be safe and have no effect on the behaviour of the protocol but may make its implementation easier to configure.

5.4.2 Per-node values

Our model allows every interface to be configured independently. However, in some protocol designs it may be valuable to have parts of policy that are shared by all interfaces on a router – for example, BGP is designed with the assumption that AS numbers are assigned to routers, and used equally for routes received or sent on any interface, and does not permit a single router to use different AS numbers over different links. This is not a requirement for convergence of the protocol to a stable routing solution, but it is an important part of the intended semantics of the protocol.

To handle per-node labels as well as per-interface labels, we could extend the configuration algebras with a new per-node label type L_N and extend the binary operators $\triangleright_E$, $\triangleright_I$ to take a third argument of type L_N to use in their computation. This would allow the language specification to define that certain fields, such as the `enum` node identifier, are to be configured once and used for all interfaces on that router.

5.4.3 Additional constraints

More complex constraints than per-node labels may be useful in some cases, to enforce the desired semantics of the protocol design and to detect erroneous configurations; the algebraic properties can only guarantee that the protocol will converge, not that it will converge to what the designers wanted. For example, our scoped product example allows any link to be arbitrarily specified as either internal or external. BGP instead ties this to the AS numbers of the routers on the link: if they are the same AS then it is an internal link (IBGP), else it is external (EBGP). This provides some guarantees of transitivity over the global network configuration (if A to B is internal, and B to C is internal, then A to C must also be internal). In a routing algorithm such as gBGP with persistent sessions between routers (unlike gRIP), it should be feasible to design a system that compares the relevant policy labels when setting up the session and aborts if they violate some arbitrary constraint. However, it is unclear exactly what (if any) types of constraint would be worthwhile in practice, and we do not explore this further.

5.4.4 Risks of shared configuration

In Section 5.3 we argued that moving computation can help performance, for example by filtering out routes before needlessly transmitting them over the network. However, there is a danger that the cost of the filtering may outweigh the benefits, particularly if a highly-connected router has many neighbours that all offload complex filters onto it and overload its processing capabilities.

To mitigate this risk, we could rely on network operators to only agree to use configuration languages with very simple forms of offloaded computation (e.g. only permit filters, and only filters that can be implemented as an efficient prefix-tree lookup or hashed attribute value lookups; don't allow large sets of arbitrary AS path regular expressions that could be very expensive to compute). We could also provide tools to estimate the worst-case computation cost of a configuration language, by analysing the generated intermediate code, to help operators make these decisions.

Chapter 6

Route maps

We have modelled the behaviour of sending a metric m over a link as an algebraic computation $m' = l \triangleright m$ where the label l is part of the network configuration, and the function $\triangleright$ is part of the routing language design. In contrast to this separation, practical routing protocol implementations support *route maps* as a way to define functions as part of the network configuration rather than as part of the routing language.

There are several deployed alternatives for route map configuration with significantly different syntax (described in the next section), but they are all based on the same fundamental model of matching a route based on a boolean combination of a pre-defined set of predicates (such as integer inequalities over metric values, or regular expression matches over AS path strings, or prefix set membership over destination addresses) and returning a route with pre-defined operations applied to its properties (such as prepending values to AS paths, or adding a constant to a metric value). Some implementations have a set of predicates and operations that are intimately tied to the details of the routing protocol and the requirements of customers, with a large range of very specialised features, while some have a more powerful and more generalised approach.

Route map implementations include restrictions to prevent some violations of protocol semantics. For example, elements typically cannot be removed from BGP AS paths (which would break BGP's loop-prevention mechanism) – the only operation is to prepend an arbitrary list of new values. However, these restrictions are not universally applied: RIP route maps are often allowed to replace the metric value with an arbitrary integer, violating the assumption that loops will always lead to an infinite metric; similarly BGP local preferences can be set to arbitrary values regardless of the commercial relationships that the local preference is typically modelling.

Route maps provide a great deal of flexibility and are a critical part of any non-trivial network configuration, but they complicate the design and analysis of a protocol. In this chapter we aim to incorporate functionality similar to route maps into the meta-

routing system without losing the ability to reason about the protocol's correctness.

First we look at the route map support provided by current protocol implementations, as an indication of the flexibility that may be important in network design, and also look at other related work in this area. We then show how to add route maps to our algebraic model, and what correctness guarantees we can retain. Finally we explore an implementation of route maps in the metarouting toolkit.

6.1 Survey of implementations

It is important to understand what the concept of "route map" is typically taken to mean, before we can consider how to implement a similar concept using metarouting. There is no standardisation in this area – routing protocol RFCs leave it as entirely implementation-defined behaviour, and every implementation takes a different approach. We are not aware of any existing comparative surveys of this topic. This section therefore gives an initial examination of the route map features and configuration syntax provided by a number of current routing protocol implementations, letting us determine the scope of their functionality and the common concepts that underlie them. This information is derived from their user documentation and from the source code of the open source implementations.

In each case we detail the general shape of route map configurations and their control flow, and the commands provided by the configuration syntax. To demonstrate that the different syntaxes provide a similar core of functionality, we express an example route map in each of the different systems. We also briefly discuss some implementation details that will be relevant when we consider how to implement route maps in metarouting.

6.1.1 Cisco IOS and Quagga

The configuration syntax design of Quagga is largely copied from Cisco IOS, so their high-level concepts and many low-level details are very similar. Our analysis will focus on Quagga, as the availability of its source code allows a more reliable understanding.

Each route map is given a distinct name, and is written as a sequence of entries. Each entry is specified with a number to determine its relative order in the sequence. An entry contains an unordered set of `match` statements, an unordered set of `set` statements, and some control flow statements. Each `match` statement is a single predicate over the route's properties; if all predicates match then the entry as a whole is considered a match.

```
match interface WORD
match ip address (<1-199>|<1300-2699>|WORD)
match ip address prefix-list WORD
match ip next-hop (<1-199>|<1300-2699>|WORD)
match ip next-hop prefix-list WORD
match metric <0-4294967295>
match tag <0-65535>
set ip next-hop A.B.C.D
set metric <+/-metric>
set metric <0-4294967295>
set tag <0-65535>
```

Figure 6.1: Quagga RIP route-map command patterns.

If an entry matches, its `set` statements will be applied. These each perform a modification to the route's properties. The entry may then call another route map (effectively a function call), continue to the next entry in the current route map's sequence, jump to any later entry in the current route map's sequence, or finish processing the route and either accept it or reject it (i.e. filtering it out). If an entry does not match, processing will continue with the next entry in the sequence.

This syntax is optimised for relatively simple cases and cannot practically handle all logic expressions, given that each entry matches on a conjunction of predicates; in general an expression can be expanded to disjunctive normal form and then written in the route map syntax, but with potential exponential expansion.

The list of available `match` and `set` commands is determined by the protocol.

Quagga RIP

Figure 6.1 shows the route map commands that Quagga supports for RIP. In addition to matching and setting the metric values, it can match on other parts of the route's properties, in particular its destination prefix and next hop. It also allows the next hop to be updated – for example if router B receives a route from A, and is going to send it to C, the default next hop will be B; but if the network operator knows that C can transmit data directly to A (despite the routing protocol not being run between C and A directly), the route map could cause the route to be advertised to C with a next hop of A.

Matching on destination prefix is a very common operation, for example to restrict a customer to only advertising routes to prefixes that are known to belong to them. The configuration language provides various ways to define sets of prefixes relatively concisely (e.g. a single command can define the set of about 2^{23} prefixes that are subsets of 10.0.0.0/8 but are not /31 or /32). We can easily reuse this mechanism when extend-

```
match as-path WORD
match community (<1-500>|WORD)
match community (<1-500>|WORD) exact-match
match extcommunity (<1-500>|WORD)
match ip address (<1-199>|<1300-2699>|WORD)
match ip address prefix-list WORD
match ip next-hop (<1-199>|<1300-2699>|WORD)
match ip next-hop prefix-list WORD
match ip route-source (<1-199>|<1300-2699>|WORD)
match ip route-source prefix-list WORD
match ipv6 address WORD
match ipv6 address prefix-list WORD
match ipv6 next-hop X:X::X:X
match metric <0-4294967295>
match origin (egp|igp|incomplete)
match pathlimit as <1-65535>
match peer (A.B.C.D|X:X::X:X)
match peer local
```

```
set aggregator as <1-65535> A.B.C.D
set as-path prepend .<1-65535>
set atomic-aggregate
set comm-list (<1-500>|WORD) delete
set community .AA:NN
set community none
set extcommunity rt .ASN:nn
set extcommunity soo .ASN:nn
set ip next-hop A.B.C.D
set ip next-hop peer-address
set ipv6 next-hop global X:X::X:X
set ipv6 next-hop local X:X::X:X
set local-preference <0-4294967295>
set metric <+/-metric>
set metric <0-4294967295>
set origin (egp|igp|incomplete)
set originator-id A.B.C.D
set pathlimit ttl <1-255>
set vpnv4 next-hop A.B.C.D
set weight <0-4294967295>
```

Figure 6.2: Quagga BGP route-map commands

ing route maps into gQuagga: all we want is a predicate that returns whether a given prefix is in a named set.

Dealing with next hops is a more complex issue since it mixes the concepts of routing and forwarding; we will return to this later.

Quagga implements route maps by defining each `match` and `set` command with a function to compile the command's argument string (typically splitting on spaces and parsing numbers) and a function to apply the compiled arguments to a route. A whole route map is compiled into a linked list of pointers to each command's compiled arguments and application function; a route is processed by iterating over the linked list and calling each function in turn.

Quagga BGP

Quagga's BGP route maps are fundamentally the same as for RIP, albeit with many more commands due to the number of route attributes that BGP provides. Figure 6.2 shows the full list.

Communities and AS paths are matched using named (or numbered) sets that are defined outside the route map, similarly to destination prefixes.

```
ip prefix-list PLIST permit 10.0.0.0/8
ip prefix-list PLIST permit 192.168.0.0/16

ip community-list CLIST permit 1234:50

ip as-path access-list ASLIST ^1235_

route-map EXAMPLE permit 10
  match ip prefix-list PLIST
  set local-preference 100
  on-match goto 30

route-map EXAMPLE deny 20

route-map EXAMPLE permit 30
  match community CLIST
  set metric +50

route-map EXAMPLE permit 40
  match as-path ASLIST
  set metric +50

route-map EXAMPLE permit 50
```

Figure 6.3: Route map example in Cisco/Quagga syntax.

AS path sets can be defined using a form of regular expressions, matching a path represented as a space-separated string of numbers. Quagga's implementation uses POSIX regular expression syntax [IEE01], with the special character "_" that matches the boundaries between numbers. For example, "^1235_" matches a path that begins with AS 1235.

Figure 6.3 demonstrates a definition of a route map named EXAMPLE. The (highly contrived) goal of this example is to accept only routes for destination prefixes in the 10.*x*.*y*.*z* and 192.168.*x*.*y* ranges, and set their local preference attribute to 100. Also, routes that either have a community list containing the community value 1234:50 or have come from a router in AS 1235 should have their MED attribute increased by 50.

In the example we first define a prefix set, community set, and AS path set. The first route map entry matches routes with the desired destination prefix, updates their local preference, then jumps to the third entry. Non-matching routes will fall through to the second entry, which has no match statements and therefore matches all routes, and is defined with deny so that matching routes will be filtered out.

The third and fourth entries are implementing the community/AS disjunction. Routes matching the community set will have their MED metric updated, and will then stop processing. Routes that fail to match will fall through to the fourth entry, updating their MED if their AS path matches the set defined by a regular expression. Routes that still fail to match will fall through to the final entry and be accepted by default.

6.1.2 JunOS

Juniper's JunOS uses different syntax but very similar concepts. A *routing policy* selects routes based on a conjunction of match conditions, and applies a set of policy terms that alter the route's attributes, with a limited form of flow control to join together multiple routing policies. Policy match conditions are typically restricted to equality with a configured constant. Policy terms typically set a value to a constant, or add or subtract a constant. The `metric` attribute (BGP's MED) is a special case, as its new value can be computed as a linear combination of the old `metric` and `metric2` (in IBGP, the IGP metric).

Figures C.1 and C.2 (in Appendix C) list the available match conditions and terms described in Juniper's documentation. Some are only applicable in a subset of protocols; `as-path` only applies to BGP, while `area` only applies to OSPF. AS path matches use regular expressions as before.

Figure 6.4 demonstrates a definition of the same example as the route map in the previous section. Instead of using `goto` as in the Cisco example, we use `default-action` in the first term to reject all routes but to continue processing. The second term overrides the action to `accept` for routes that match the destination prefix set, and updates their local preference. The third and fourth terms update the metric when appropriate, and perform an immediate `accept` which will stop the processing of the route map. (This is necessary so that a route matching both the third and fourth terms will not have its metric increased twice.)

6.1.3 Cisco IOS XR

Whereas the traditional Cisco IOS uses the syntax described earlier (copied by Quagga), the newer Cisco IOS XR system has a redesigned approach based on its Routing Policy Language (RPL), with the aim of supporting more modular configuration and more complex logic.

RPL is a simple language with basic control flow statements (`if`, `else`, `elseif`, `and`, `or`, `not`). It defines expressions to check particular route attributes, such as the form

```
local-preference {eq | is | ge | le} number
```

```
policy-options {
  prefix-list PLIST { 10.0.0.0/8; 192.168.0.0/16; }
  community CLIST members 1234:50;
  as-path ASLIST name "^1235_";

  policy-statement example {
    term set-default {
      then default-action reject;
    }
    term accept-plist {
      from prefix-list PLIST;
      then {
        local-preference 100;
        default-action accept;
      }
    }
    term update-metric-clist {
      from community CLIST;
      then {
        metric add 50;
        accept;
      }
    }
    term update-metric-aslist {
      from as-path ASLIST;
      then {
        metric add 50;
        accept;
      }
    }
  }
}
```

Figure 6.4: Route map example in JunOS syntax.

(variables in these command definitions are indicated in italics, choices are indicated with the syntax `{ A | B }`; in this case the operators correspond to equality, equality (again), greater-than-or-equal, less-than-or-equal).

It also provides a fairly inflexible set of commands to alter the route attributes, such as

`set local-preference` *number*

`set med {`*number* `| igp-cost | {+ | -}` *number* `| max-reachable}`

```
route-policy example
  if destination in (10.0.0.0/8 192.168.0.0/16) then
    set local-preference 100
    if community matches-any (1234:50) or as-path neighbor-is '1235' then
      set med + 50
    end
  else
    drop
  end
end-policy
```

Figure 6.5: Route map example in Cisco IOS XR syntax.

These commands are executed sequentially, and can override or alter values set by earlier commands.

Functionally this is very similar to the traditional Cisco route maps, though with more powerful syntax and without the need to express complex logic in disjunctive normal form.

Figure 6.5 implements the previous route map example in the IOS XR syntax. The control flow is much clearer in this version, due to the ability to nest expressions and to use "`or`" to combine conditions. The sets of prefixes and communities are written inline here, but could be defined outside of the `route-policy` if they are long or are shared by multiple policies. AS path regular expressions are still supported, but the `as-path neighbor-is` expression is a more readable way to achieve the same behaviour.

6.1.4 BIRD

BIRD [bir] has a more powerful and more generic approach. It lists as a design goal:

> *Offer powerful route filtering.* There already were several attempts to incorporate route filters to a dynamic router, but most of them have used simple sequences of filtering rules which were very inflexible and hard to use for non-trivial filters. We've decided to employ a simple loop-free programming language having access to all the route attributes and being able to modify the most of them.

It uses *filters* that are written in a C-like language with control flow (`if`, `case` switches, but no loops) and local variables and a type system. The types are listed in Figure 6.6. There are a number of operations that can be performed on types: arithmetic and comparisons on integers; logical operations on booleans; membership on sets; masking on

Type	Example literals
`bool`	`true`
`int`	`1234`
`pair`	`(1234, 5678)` or `(1+2, a)`
`quad`	`127.0.0.1`
`string`	`"BGP"`
`ip`	`10.20.30.40` or `fec0:3:4::1`
`prefix`	`192.168.0.0/16`
`int|pair|quad|ip|enum set`	`[ 1, 2, 5..7 ]` or `[ (111, 222), (123, *) ]`
`prefix set`	`[ 1.0.0.0/8, 2.0.0.0/8{16,24} ]`
`enum`	`ORIGIN_IGP`
`bgppath`	*no literals*
`bgpmask`	`[= 100 * =]` or `[= * 4 (1+2) a =]`
`clist`	*no literals*

Figure 6.6: BIRD filter data types.

`ip`; the functions `P.first`, `P.last`, `P.len` and `prepend(P, A)` on `bgpath`; the functions `add(C, P)` and `delete(C, P)` on `clist`; and matching paths against masks.

When a filter is executed for a particular route, a number of route attributes are exposed as variables to the filter code for reading and (in most cases) writing, as in Figure 6.7. BIRD parses the configuration into an AST and executes filters using an interpreter.

The combination of primitive types that are tailored to the routing environment plus a small set of operations that can apply to any route attributes of the right type (instead of ad hoc decisions to allow arithmetic on MED but not on local preference, for example) make BIRD's filter language much more expressive than the approaches discussed earlier, while also being easier to define and to learn.

Figure 6.8 repeats the route map example in the BIRD syntax.

6.1.5 XORP

Bittau and Handley [BH06] describe the design of XORP's policy system. A *policy statement* consists of a list of *policy terms*. Each term specifies a conjunction of *match conditions*, and each match condition is a boolean operator comparing a route attribute to a constant value (for example "`localpref < 100`"). If a route matches all the conditions in a term, the term's *actions* are applied. Actions can modify a route's attributes with an operator and a value (such as "`localpref = 100`" or "`med add 50`"), and can also accept or reject routes and jump to other terms. Policies can also call other policies as subroutines.

The XORP policy manager checks the configuration for validity, and then compiles

Type	Name	Notes
Common route attributes:		
prefix	net	read-only
enum	scope	
int	preference	
ip	from	read-only
ip	gw	
string	proto	read-only
enum	source	
enum	cast	read-only
enum	dest	read-only
RIP route attributes:		
int	rip_metric	
int	rip_tag	
BGP route attributes:		
bgppath	bgp_path	
int	bgp_local_pref	
int	bgp_med	optional
enum	bgp_origin	ORIGIN_IGP|ORIGIN_EGP|ORIGIN_INCOMPLETE
ip	bgp_next_hop	
void	bgp_atomic_aggr	optional
clist	bgp_community	optional

Figure 6.7: BIRD filter route attributes.

```
filter example
prefix set PLIST;
{
  PLIST = [ 10.0.0.0/8, 192.168.0.0/16 ];

  if !(net ~ PLIST)) then
    reject "unwanted prefix";

  bgp_local_pref = 100;

  if ((1234,50) ~ bgp_community || bgp_path.first = 1235) then
    bgp_med = bgp_med + 50;

  accept;
}
```

Figure 6.8: Route map example in BIRD syntax.

it into a simple stack machine assembly language which is executed by the back-end route processing code.

This statement/term/condition/action design is similar in structure to the Cisco IOS and JunOS approaches, and the syntax is superficially very similar to JunOS, but the implementation has a level of generality closer to BIRD due to defining conditions and actions over datatypes instead of over specific route attributes. Figure C.3 is part of XORP's definition of configuration syntax for BGP policy: the `from` values are import conditions, the `to` values are export conditions, and the `then` values are actions that can be used for both import and export.

The definition in Figure C.3 is not a complete specification as some attributes have more specific types in XORP's policy backend. For example, `community` is `txt` (an arbitrary string) in the configuration syntax, but in the backend is represented with the C++ type `ElemSetAny<ElemCom32>` (a set type specialised for elements of type `ElemCom32` which is effectively a 32-bit integer with special parsing/printing functions to match the conventional BGP community string format). Unfortunately it is non-trivial to match the configuration syntax to the backend types, as they are separated by an RPC mechanism that ignores types and represents all values as strings, and the backend relies on run-time polymorphism to determine the type of each attribute, so there is no declarative specification of the attribute types.

Figure 6.9 implements the earlier route map example in XORP's configuration syntax. The first term in the `example` statement includes the `example-accept-plist` statement as a subroutine, to reject routes that do not match the prefix list. The second term uses "`community >= "1234:50"`" to perform a superset test (the string is split on spaces to give (in this case) a single community value 1234:50).

6.2 Configuration template languages

Configuring routers in a large-scale network is often a major manageability challenge. A number of tools have been developed that attempt to address the problem by adding a layer on top of the router's native configuration language (typically the traditional Cisco IOS language). This problem is outside our scope but it is relevant in its interaction with the configuration languages.

6.2.1 RPSL

RPSL [AVG+99], first published in 1998, was designed as a common vendor-independent language for expressing BGP routing policies, allowing them to be stored in a global database and used for analysis or for automating parts of network configuration. The

```
network4-list plist {
  network 10.0.0.0/8
  network 192.168.0.0/16
}
policy-statement example-accept-plist {
  term a {
    from {
      network4-list: "plist"
    }
    then { accept }
  }
  term b {
    then { reject }
  }
}
policy-statement example {
  term a {
    from {
      policy: "example-accept-plist"
    }
    then {
      localpref = 100
    }
  }
  term b {
    from {
      community >= "1234:50"
    }
    then {
      med add 50
      accept
    }
  }
  term c {
    from {
      as-path: "^1235( |$)"
    }
    then {
      med add 50
      accept
    }
  }
}
```

Figure 6.9: Route map example in XORP syntax.

integer[lower, upper]	email	filter
real[lower, upper]	as_number	as_set_name
enum[name, name, ...]	ipv4_address	route_set_name
string	address_prefix	rtr_set_name
boolean	address_prefix_range	filter_set_name
rpsl_word	dns_name	peering_set_name
free_text		

Figure 6.10: RPSL predefined types.

RtConfig tool [MSO+99] can compile RPSL into Cisco IOS or JunOS configuration syntax.

The RPSL language defines a *filter-set* which identifies routes by a boolean combination of conditions. Conditions consist of destination prefix matches, AS path regular expressions, and *RSPL dictionary* expressions. Policy specifications can be defined on import and export, matching routes from a given peer matching a given filter, applying a sequence of RPSL dictionary expressions. The RPSL dictionary is an extensible model of BGP route attributes and other protocol or router attributes. Attributes in the dictionary are not typed; instead they each define an attribute-specific list of available operators. The operator arguments (not counting the attribute itself) have one of the predefined types in Figure 6.10, or a list or union or dictionary over those types. For example, the only operator defined for the `pref` attribute is

```
operator=(integer[0, 65535])
```

which assigns a constant integer. The `med` attribute, which is conceptually also an integer, instead has the operator

```
operator=(union integer[0, 65535], enum[igp_cost])
```

which allows the RPSL expressions `med = 10` and `med = igp_cost` (i.e. assigning the value of a different attribute). The only filter expression defined in RFC 2622 is `contains` on the `community` attribute.

In practice, most of the features of RPSL appear to be little used in its primary usage in Internet Routing Registries (IRRs). For example, a snapshot of the RIPE IRR database[1] has 17741 `aut-num` objects (each corresponding to a BGP AS); of these, 82 use the `community` attribute in their specified import/export actions, and 25 use it in import/export filters. Meanwhile, 395 mention "community" or "communities"

[1] `ftp://ftp.ripe.net/ripe/dbase/`

in plain text comments in a `remarks` section. The databases published by other IRRs show similar behaviour. Separately, analysing a BGP routing table dump[2] shows 32007 distinct AS numbers in AS paths, and community values associated with 1710 ASes. While these are very rough measures, they indicate a very small proportion of ASes publish their configuration to IRRs with enough detail to model how they handle BGP community attributes, and a majority of those use manually-maintained non-machine-readable text instead of RPSL.

6.2.2 Other research

While RPSL is still in relatively widespread usage in IRRs, there have been many attempts to develop new approaches to high-level network configuration.

Böhm et al. [BFM+05] present a system that allows a BGP network layout and network-wide routing policy to be specified as a series of XML files. Vendor-specific configuration file fragments (typically route map commands) are specified in a simple templating language, and the system then combines fragments based on the network and policy specifications to produce the individual router configuration files.

PRESTO [EMG+07] implements a more extensive system, to include aspects of router configuration beyond BGP (in particular for configuring VPNs). It provides a complex templating language that manipulates fragments of configuration text, driven by SQL-like queries on a relational database that describes the network design.

Nettle [VH09] implements a domain-specific embedded language in Haskell for declaratively expressing routing configuration. This is then compiled into the low-level router configuration syntax. The use of Haskell as a host language allows new abstractions to be defined by users, with safety guaranteed by Haskell's type checker as long as the standard Nettle configuration-compiler is correct. This contrasts with the templating approaches where users are expected to write configuration syntax fragments that can easily introduce syntax errors.

These approaches to high-level policy specification are all layered on top of the low-level router configuration syntax, which creates challenges with extensibility, expressivity, complexity, and correctness. Our aim is to redesign the low-level configuration as part of the metarouting system so that it is easily extensible to new protocols and new types of policy, and so that it has well-defined semantics and verifiable correctness requirements. This would complement the high-level policy systems by providing a stronger and safer foundation on which they can build, circumventing the accidental complexity in the current ad-hoc route map configuration syntaxes.

[2] `http://archive2.routeviews.org/bgpdata/2009.07/RIBS/rib.20090701.0000.bz2`

6.3 Route maps in algebraic routing

Having investigated the scope of route maps as currently implemented in routing protocols, we will now consider how to reproduce a close approximation of its behaviour algebraically.

Our original algebraic model of routing defines route computation as

$$m' = l \triangleright m.$$

We want to extend this to mirror the capabilities of route maps by encoding some form of function in the network configuration, but still maintain guarantees about the algebraic properties of the computation, in particular that it is still increasing ($m \prec m'$) and therefore safe to use in DBF algorithms. We could perform the route computation as

$$m' = f(d,\ m)$$

where d is the destination prefix (from set of all prefixes D) and f is an arbitrary route map function determined by the network configuration (specified in some kind of domain-specific programming language), as with the route map implementations we have examined. However, we would have no way to automatically verify the algebraic properties, given the general infeasibility of proving mathematical statements about programs unless the language is extremely restricted. Instead of restricting the language of f, we choose a slightly more constrained formulation of route computation,

$$m' = f(d,\ m) \triangleright m$$

based on an underlying routing algebra $(S,\ L,\ \preceq,\ \triangleright)$. Now we don't need to worry about what value f returns – we merely need it to return a value of the label type L, and our existing understanding of $\triangleright$ allows us to determine the algebraic properties. The language in which f is implemented must have a type system capable of ensuring the return value is type L, and it should avoid any non-terminating computation, but otherwise the language design is free to focus on expressivity and usability with no risk of compromising safety.

This new route computation is quite different to the original form that has been used in proofs of algorithm behaviour, as the first operand of $\triangleright$ is no longer a constant label and now depends on m. For simplicity we would like to reuse those existing proofs instead of rewriting them with the new form of computation, and we can do this by showing that any routing language in this new form is equivalent to a routing algebra in the original model.

First, note that every destination prefix is handled entirely independently by the routing algorithms – we usually don't refer to the destination at all when discussing the

algorithms. This means we can assume a fixed value of d, and then repeat the proof for every possible d. Given an arbitrary route map function f, let f_d be a curried function such that $f_d(m) = f(d, m)$. Let $\#f_d$ be a symbolic representation of that function. (This is always possible if f is implementable on a Turing machine, which is not a significant practical restriction). Now define $\widetilde{\triangleright}$ such that

$$\#f_d \mathbin{\widetilde{\triangleright}} m \equiv f(d, m) \triangleright m$$

for all f, m. There is a one-to-many relationship between f and $\#f_d$ (i.e. no distinct f_1, f_2 can have the same symbolic representation) so this is well-defined.

We now have a routing algebra $(S, \widetilde{L}, \preceq, \widetilde{\triangleright})$ where $\widetilde{L}$ is the set of symbolic representations of all computable functions of type $S \to L$. Conceptually, this $\widetilde{\triangleright}$ acts like an interpreter for the program described by $\#f_d$ – it is a perfectly normal routing algebra but one that is far too complex to define with the standard metarouting constructors (especially in terms of property inference), necessitating this new approach based on route maps. We already know that $\triangleright$ is an increasing function, so

$$m \prec f(d, m) \triangleright m$$

and the definition of $\widetilde{\triangleright}$ results in

$$m \prec \#f_d \mathbin{\widetilde{\triangleright}} m$$

and this new routing algebra is also increasing. We can therefore use this algebra safely in vector routing algorithms using the traditional form of route computation $m' = \widetilde{l} \mathbin{\widetilde{\triangleright}} m$; and since this is equivalent to $m' = f(d, m) \triangleright m$ by construction, we can equivalently implement the algorithms with the route map computation $f(d, m)$ instead of implementing the impractically complex $\widetilde{\triangleright}$.

However, not all algebraic properties are preserved: if the routing algebra based on $\triangleright$ was distributive, we do *not* have a guarantee that $\widetilde{\triangleright}$ will be distributive. The next section discusses this in more detail.

Note that this proof works because f is applied only to a value d that we can assume is a global constant, and to the metric m. If we had other inputs to f, such as the time of day or earlier routing state or a random number generator, we would no longer be able to show that it acts the same as a basic routing algebra (and, indeed, it would be easy to define an f such that the network will never converge).

Referring back to the previous analysis of current route map implementations, matching on next hop is the only common condition that is not supported by this algebraic model of route maps; we treat this as forwarding information that is handled entirely by the routing algorithm, not by the routing language. However, similar functionality could be achieved by modifying the routing language to be a lexicographic product

with a new identifier attribute using the `left` constructor (so that it is replaced on each link) with no effect on the ordering of routes, so that the language encodes the next-hop data directly. Some other match conditions in current implementations, in particular matching the network interface a route was received on or is being sent to, have no effect on expressivity (it is always possible to split them into a separate route map per interface). In general, this model of f as a deterministic function over d and m is sufficient for emulating the behaviour of current implementations.

6.3.1 Preserving distributivity

Whereas implementing route maps with an unrestricted f can preserve the increasing property of a routing algebra, it cannot preserve distributivity in general. But would it be feasible to have a more restricted form of functions that would preserve distributivity? Given an algebra $(S,\ L,\ \preceq,\ \triangleright)$ with

$$m_1 \preceq m_2 \Rightarrow l \triangleright m_1 \preceq l \triangleright m_2$$

for all $m_1, m_2 \in S, l \in L$, we want to define a subset $F' \subseteq (S \to L)$ such that

$$m_1 \preceq m_2 \Rightarrow f_d(m_1) \triangleright m_1 \preceq f_d(m_2) \triangleright m_2$$

for all $f_d \in F'$.

One immediate restriction is that these distributive route maps cannot be used for arbitrary route filtering. For example, consider the algebra $(\mathbb{N}^\infty,\ \mathbb{N}^\infty,\ \leq,\ +)$ and the function

$$f_d(m) = \begin{cases} \infty & \text{if } m = 15 \\ 1 & \text{otherwise} \end{cases}$$

which filters out routes with some particular metric by forcing them to ∞. This violates distributivity because $15 < 16$ but $f_d(15) \triangleright 15 = \infty\ >\ f_d(16) \triangleright 16 = 17$.

However, the function

$$f_d(m) = \begin{cases} \infty & \text{if } 15 \leq m \\ 1 & \text{otherwise} \end{cases}$$

does preserve distributivity. (This emulates the behaviour of a bounded-integer algebra.)

This suggests a general mechanism for constructing a distributivity-preserving subset F' that can use predicates over destinations and metrics. Assume $L \subseteq S$ for simplicity. Start with the constant functions $c_l(m) = l$. Now pick predicates such that $P(d, m_1) \wedge m_1 \preceq m_2 \Rightarrow P(d, m_2)$, and define the functions

$$g_{l,P,f}(m) = \begin{cases} l \triangleright f(m) & \text{if } P(d, m) \\ f(m) & \text{otherwise} \end{cases}$$

where $f \in F'$, with F' defined recursively to contain all c and g functions. With this approach, route maps can select routes based on any P and add any cost l to them, and the recursive definition lets this be repeated any number of times.

This seems to allow the programmable flexibility that route maps are designed for, but the significant problem is the restriction on P. It can perform arbitrary matching on d, but for a given d it is very limited. If $\preceq$ is a total order, any possible P_d is equivalent to $P_d(m) = n \preceq m$ for some $n \in S$; if $\preceq$ is a partial order, any P_d is a disjunction of these. The most common design of a routing algebra is as a lexicographic product, but in such an algebra with $S = A \times B$, no P could depend solely on B – if it matches (a_1, b_1) then it will have to match (a_2, b_2) when $a_1 \preceq a_2$, which means we lose the power to select routes based on any field but the first.

Because of these restrictions, we consider distributivity to be too onerous a requirement for any practical definition of route maps, so we will not consider it further in our design.

6.4 Route maps in metarouting

We have shown that we can safely configure a network using functions $f : D \times S \to L$ on arcs, instead of static labels $l : L$. Now we face the question of how to practically define f, when the rest of the algebra is defined with the metarouting system.

The earlier analysis of routing protocol implementations, especially XORP and BIRD, gives confidence that the approach of assigning types to route attributes and then defining general operators over types, as opposed to defining specific operators for each attribute individually (as with the older designs of Cisco, JunOS and RPSL), is feasible to implement efficiently and usable by network operators for protocols with the complexity of BGP. As the developers of XORP state, "it appears that a small set of operators and types can handle all the policies supported by commercial router vendors, for all the main routing protocols" [BH06].

The type systems in these implementations are broadly similar to metarouting's ERL types. The main difference is that they treat routes as having a flat list of named attributes, whereas ERL may have more complex nested record types. In Section 5.2.3 we described a method of constructing flat string identifiers for values in nested types (in that case for configuration annotations), so the difference is merely syntactic. It therefore seems natural to use ERL types directly for the arguments and return type of f when implementing route maps in metarouting; there is no need to introduce an extra layer of abstraction to translate the ERL types into a different set of types before operating on them.

To implement these functions over ERL types, we could use any kind of side-effect-free

programming language, for example a functional language like Haskell, or a C-like language (as with BIRD), or a simplistic new language specialised for writing route maps (as with Cisco IOS XR). The main difficulty is that the ERL type L may not fit cleanly within the type systems of these languages. For example L could be a bounded integer type with the range $[1, 100]$, while most languages only provide arithmetic on unbounded or n-bit integers. The type S is less problematic: the type system is likely to support some superset of any given ERL type, for example a 32-bit integer type instead of $[1, 100]$, so functions may safely use that as the argument type; but the return value must be in L and not just a superset. We could have the user's code return values in a superset (e.g. any 32-bit integer) and wrap the code in a verification layer that reports an error and returns a default value (typically infinity) if the value is not in L (in this case by testing $1 \leq l \leq 100$), but that would be an inelegant approach and the reliance on run-time type checking would make route maps more error-prone than if they had compile-time checking.

A more robust solution would be to replace the language's integer types and operations with some custom types (corresponding precisely to ERL types) and custom operations that make out-of-range values impossible. For example, instead of providing a function that sums values corresponding to the ERL type `TyIntRange(1, 100)` and would have to fail on overflow, provide only a function that sums values of type `TyAddConst(W, TyIntRange(1, 100))` so it can always return a valid value of that type (using `W` as a sensible result on overflow). This would involve implementing the full range of ERL's types in the chosen language. In principle we could do this for any language, and in fact we already have two languages that implement these types: C++ (via *libmrc*), and ERL itself. These are far from ideal languages for this purpose – C++ provides no safety guarantees and is difficult to compile, while ERL was designed as an intermediate language not to be used directly by humans, so preferably we would replace it with a more usable language syntax – but they are sufficient for demonstrating the fundamental concepts, and designing yet another language would be an unnecessary distraction for this work. We will therefore use ERL as the language for exploring this approach to implementing route maps.

Previously we have only put ERL constant values in configuration files, and all other ERL expressions have been compiled to C++ and then to machine code in an offline process. Now we want to write route maps as ERL transform expressions, and there are a number of ways we could implement this. Firstly, we could embed an ERL parser and interpreter inside the routing algorithm implementation. (The interpreter could run on the parsed AST as in BIRD, or on a stack machine bytecode format as in XORP.) Alternatively, we could use the existing C++ compiler infrastructure – either running on the router, or delegated to a remote server if the router has insufficient resources to run the compiler directly – to compile the route maps into efficient machine code whenever the configuration changes. These are standard programming language tech-

niques, and different implementations may choose different tradeoffs. Similar options would be available if route maps were written in languages other than ERL: the important factor is that we have a well-defined functional language that permits this flexibility without affecting the semantics of the protocol.

ERL already has most of the types that we need for route maps, and the operations for constructing values of those types, but our earlier analysis of BGP route map syntaxes indicates that we need to extend it with more predicates to handle the route-matching features that route maps typically provide. The only types we will add are `TyPrefix` and `TyPrefixSet`, for handling IP address prefixes, and a binary operator `BoPrefixIn(`p`:TyPrefix, `s`:TyPrefixSet):TyBool`. In gQuagga we could implement `TyPrefixSet` as a reference to a named prefix-list that is defined using Quagga's standard prefix-list mechanisms, so there is no need to define new syntax for parsing prefixes. The other expressions we add for completeness are an equality operator, and comparison operators over a preorder:

ERL syntax	Semantics
`BoEq(`a`:`T`, `b`:`T`) : TyBool`	$[\![a]\!] = [\![b]\!]$
`BoLte(`p`:`T`, `a`:`T`, `b`:`T`) : TyBool`	$[\![a]\!][\![p]\!][\![b]\!]$
`BoLt(`p`:`T`, `a`:`T`, `b`:`T`) : TyBool`	$[\![a]\!][\![p]\!][\![b]\!] \wedge \neg([\![b]\!][\![p]\!][\![a]\!])$

Many route map behaviours can be expressed using more general features of ERL, so we do not add special cases for them all. For example, community lists can be implemented as type `TySet(TyInt)` with matching based on set intersection: the ERL expression

```
UoBoolNot(UoSetEmpty(SgSetInter(c, ExprSet(100, 200))))
```

corresponding to the algebraic expression

$$\neg(|c \cap \{100, 200\}| = 0)$$

will match a set `c` that contains either 100 or 200. The C++ implementation of this expression will be reasonably efficient, as sets are stored as red-black trees and so intersection has cost $O(n + m)$ in the sizes of the sets. The constant factors could be reduced by adding special-casing in the ERL-to-C++ compiler to return after the first match when an expression of the form `UoSetEmpty(SgSetInter(...))` is seen. Exact community matches can simply use `BoEq`, while new values can be added with `SgSetUnion` and removed with `SgSetDiff`.

The power of this approach is that it will work for more complex ERL types, such as sets of records that contain a pair of integers (as with BGP communities), or sets over a disjoint union of various differently-typed complex values (as with BGP extended communities): the small collection of primitives supplied by ERL is sufficient for a

wide range of behaviour over a wide range of routing languages, and it is not necessary to hard-code the implementation of every new match condition. A higher-level syntax, with shortcuts for writing these kinds of community expressions at a higher level of abstraction, would make this system more usable without losing the power of the underlying programmable model.

6.5 Examples

To reproduce the route map example of Section 6.1 in the metarouting system, we will first define a (partial) routing language with the relevant attributes for an extremely simplified version of BGP:

```
let type comm =
  TyRecord(
    as = TyIntRange(0, 65535),
    local = TyIntRange(0, 65535)
  )

let type sig =
  TyAddConst(INF,
    TyRecord(
      aspath = TyListSimp(TyIntRange(0, 65535)),
      communities = TySet(comm),
      localpref = TyIntNonNeg,
      med = TyIntNonNeg
    )
  )

let type lbl =
  TyAddConst(INF,
    TyRecord(
      asnum = TyIntRange(0, 65535),
      communities = TySet(comm),
      localpref = TyIntNonNeg,
      med = TyIntNonNeg
    )
  )
```

We can then implement the route map as an ERL transform expression, shown in Figure 6.11. This route map expression can be compiled with mrc into C++ code, then loaded into a gQuagga routing protocol with an extension to the mrc API (including

```
let transform rmap = fun (d:TyPrefix s:sig) ->
  ExprCond(
    (* Check the prefix is in the TyPrefixSet named 'plist' *)
    BoPrefixIn(d, "plist"),

    (* If so, return a new label record *)
    TyRecord(
      asnum = 1234,
      communities = ExprSelect(s, communities),
      localpref = 100,
      (* Update med depending on complex condition *)
      med = ExprCond(
        BoOr(
          (* Either the community 1234:50 is in the communities list ... *)
          UoBoolNot(UoSetEmpty(SgSetInter(
            ExprSelect(s, communities),
            ExprSet(ExprRecord(as=1234, local=50) : comm)
          ))),
          (* ... or the aspath starts with 1235 *)
          BoEq(
            UoListHead(ExprSelect(s, aspath)),
            1235
          )
        ),

        (* Add 50 to previous med value *)
        ExprBinop(BoSemigroup(SgIntPlus), ExprSelect(s, med), 50),

        (* Otherwise pass through med unchanged *)
        ExprSelect(s, med)
      )
    )

    (* Prefix d not in plist, so return infinity to reject the route *)
    INF
  )
```

Figure 6.11: Route map implemented as ERL transform expression.

a callback function so the compiled `BoPrefixIn` operator can perform a lookup using gQuagga's standard prefix lists).

6.6 Conclusions

Earlier work with XORP and BIRD has shown that it is possible to implement route maps for protocols such as BGP using a small set of common types and operations, providing greater power than the inflexible specialised behaviours implemented by many other router implementations. We have shown that this can be extended to any protocol defined by a routing algebra, by expressing the route maps in a language similar to ERL, and further that we can add correctness guarantees to route maps based on algebraic properties.

Chapter 7

Case study

Figure 7.1 reprises the implementation part of the system diagram from Chapter 1; this dissertation has now explored most of the components. In Chapter 3 we defined the RAML and ERL languages, and the translation from RAML to ERL. The compilation from ERL to C++ is described in other work [Bil09]. Chapter 4 covered the implementation of gQuagga routing algorithms and the linkage with a compiled routing language. Chapter 5 (interface configuration) and Chapter 6 (route maps) extended the routing languages and algorithms with important features for vector protocols.

In this chapter we collect all the components together into a working protocol implementation, to demonstrate how they interact and how the completed system can be used.

7.1 Scoped product protocol

In Section 3.3.3 we introduced the scoped product algebra, which implements a network model split into regions with links separated into intra-region (internal) and inter-region (external). This is not intended to be a practical design for a routing protocol – it is too limited to support much expressive policy, and the region model may be unnecessarily complex for small networks and insufficiently scalable for large networks – but it demonstrates the major principles and features of the metarouting system and the general shape of potential future routing language designs.

In Section 5.2 we extended the RAML specification of the algebra to support separate export and import interface labels. The definition of RAML lets us convert the algebra into an ERL implementation, which can then be compiled into C++. (We do not have a sufficiently complete RAML-to-ERL compiler at the time of writing, so for this case study we wrote the equivalent ERL code by hand.) Now we can link that C++ code to the algorithm implementation and produce a usable routing protocol.

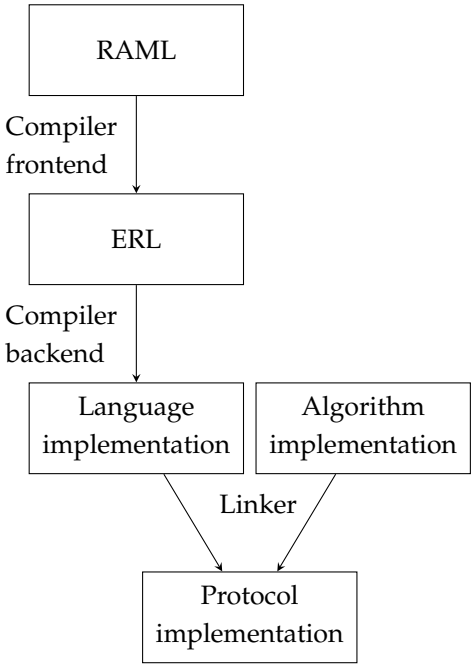


Figure 7.1: Components of the implementation of the metarouting toolkit.

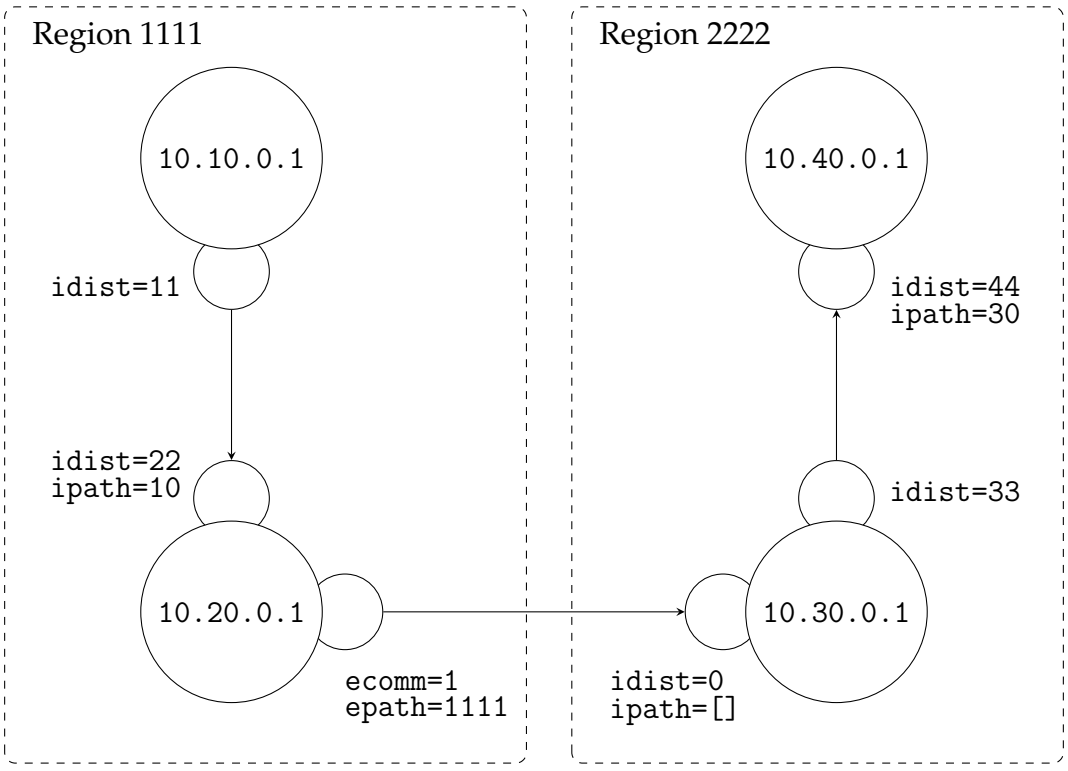


Figure 7.2: Network with interface labels from the scoped product routing language.

```
In mode "route-policy NAME out":
  set external ecomm <0-2>
  set external epath <0-2147483647>
  set external idist unit
  set external ipath unit
  set internal ecomm unit
  set internal epath unit
  set internal idist <0-2147483647>
  set internal ipath unit
  set w
```

```
In mode "route-policy NAME in":
  set external ecomm unit
  set external epath unit
  set external idist <0-2147483647>
  set external ipath INDEX <0-2147483647>
  set external ipath empty
  set external ipath notsimple
  set internal ecomm unit
  set internal epath unit
  set internal idist <0-2147483647>
  set internal ipath <0-2147483647>
  set w
```

Figure 7.3: gQuagga configuration command templates for specifying export and import policy labels.

```
bgpd(config)# route-policy example in
bgpd(config-route-policy-in)# set ?
  w         Set value to constant
  external  Set external value
  internal  Set internal value
bgpd(config-route-policy-in)# set external ?
  ecomm  Set ecomm value
  epath  Set epath value
  idist  Set idist value
  ipath  Set ipath value
bgpd(config-route-policy-in)# set external ipath ?
  notsimple  Set value to constant
  empty      Set to empty list
  INDEX      Set nth (starting from 0) component of list
bgpd(config-route-policy-in)# set external ipath 0 ?
  <0-2147483647>  Set bounded integer value
bgpd(config-route-policy-in)# set external ipath 0 1234
bgpd(config-route-policy-in)#
```

Figure 7.4: Output of interactive gBGP configuration shell.

Figure 7.2 illustrates the network configuration we will use for this example, with the relevant fields of policy labels listed next to each interface. For simplicity we only include links in a single direction, allowing routing information to flow from router `10.10.0.1` through to router `10.40.0.1`; links in the other direction will be labelled with `W`.

As discussed in Section 4.5.2, we automatically generate gQuagga configuration command templates from the ERL code. Figure 7.3 lists the commands from this example.

```
BGP: 10.10.0.1 rcvd UPDATE w/ attr: nexthop 10.10.0.1,
  mr-metric inject(internal, <ecomm = 0, epath = [], idist = 11, ipath = []>)
BGP: 10.10.0.1 rcvd 10.10.0.1/32
BGP: 10.30.0.1 [FSM] Timer (routeadv timer expire)
BGP: 10.30.0.1 send UPDATE 10.20.0.1/32
BGP: 10.30.0.1 send UPDATE 10.10.0.1/32
BGP: 10.30.0.1 rcvd UPDATE w/ attr: nexthop 10.30.0.1, mr-metric W
BGP: 10.30.0.1 rcvd UPDATE about 10.30.0.1/32 -- DENIED due to: metric is infinity;
```

Figure 7.5: Fragment of output from gBGP process running on router 10.20.0.1.

Figure 7.4 shows the output of gQuagga's interactive configuration shell when linked with this routing language: entering a partial command followed by a "`?`" prints a list of the tokens that are allowed to come next in the command. Each command sets either the whole label to a given value (e.g. "`set w`"), or a component of the nested record/union structures, or an element of a list/set component (e.g. "`set external ipath INDEX ...`" to set the `INDEX`th element). Once a set of commands has been given, followed by an "`end-policy`" command, the policy is checked for correctness: for example if any "`set external ...`" command is used then the label cannot also be set to `w` or an `internal` value, and if it is an external export policy then both `ecomm` and `epath` must be specified, else an error is reported and the policy is rejected. (Fields of a record which are of unit type, and can only take the value `unit`, are allowed to be omitted for convenience.)

Figure 7.6 lists the configuration files necessary for running this network with gBGP. The "`bgp router-id`" lines define the ID of the router with that configuration file. The "`network`" lines list IPv4 address prefixes which will be originated as new routes by the router. (In this example we make each router advertise its own ID as an IP address.) The "`default-metric`" blocks specify the metric value that will be assigned to each of these originated routes.

The "`neighbor`" lines indicate that the router should set up a BGP session to the named router, using the named `route-policy` label on the export ("`out`") or import ("`in`") interface of the new link. These command refer to the"`route-policy`" blocks that define export labels or import labels using the commands described earlier.

We can now run this network with gBGP and observe its behaviour. Figure 7.5 shows a part of the debug log output of router 10.20.0.1: first it receives a route from 10.10.0.1, whose wire metric is tagged as internal and with an `idist` of 11, corresponding to the export policy of 10.10.0.1. Next it sends this route, plus its own originated route 10.20.0.1/32, to router 10.30.0.1. Finally it receives a route back from 10.30.0.1, with wire metric `W` (due to the policies we configured for the 'reverse' direction around the network), which is denied (i.e. not inserted into the routing table and not used for routing any traffic) because it is an infinity value.

```
router bgp
 bgp router-id 10.10.0.1
 network 10.10.0.1/32
 neighbor 10.20.0.1 route-policy 10-to-20 out
 neighbor 10.20.0.1 route-policy 20-to-10 in

default-metric
 set ecomm 0
 set epath empty
 set idist 0
 set ipath empty
end-metric

route-policy 10-to-20 out
 set internal idist 11
end-policy

route-policy 20-to-10 in
 set w
end-policy
```

```
router bgp
 bgp router-id 10.40.0.1
 network 10.40.0.1/32
 neighbor 10.30.0.1 route-policy 40-to-30 out
 neighbor 10.30.0.1 route-policy 30-to-40 in

default-metric
 set ecomm 0
 set epath empty
 set idist 0
 set ipath empty
end-metric

route-policy 30-to-40 in
 set internal idist 44
 set internal ipath 30
end-policy

route-policy 40-to-30 out
 set w
end-policy
```

```
router bgp
 bgp router-id 10.20.0.1
 network 10.20.0.1/32
 neighbor 10.10.0.1 route-policy 20-to-10 out
 neighbor 10.10.0.1 route-policy 10-to-20 in
 neighbor 10.30.0.1 route-policy 20-to-30 out
 neighbor 10.30.0.1 route-policy 30-to-20 in

default-metric
 set ecomm 1
 set epath empty
 set idist 0
 set ipath empty
end-metric

route-policy 10-to-20 in
 set internal idist 22
 set internal ipath 10
end-policy

route-policy 20-to-30 out
 set external ecomm 1
 set external epath 1111
end-policy

route-policy 20-to-10 out
 set w
end-policy

route-policy 30-to-20 in
 set w
end-policy
```

```
router bgp
 bgp router-id 10.30.0.1
 network 10.30.0.1/32
 neighbor 10.20.0.1 route-policy 30-to-20 out
 neighbor 10.20.0.1 route-policy 20-to-30 in
 neighbor 10.40.0.1 route-policy 30-to-40 out
 neighbor 10.40.0.1 route-policy 40-to-30 in

default-metric
 set ecomm 0
 set epath empty
 set idist 0
 set ipath empty
end-metric

route-policy 20-to-30 in
 set external idist 0
 set external ipath empty
end-policy

route-policy 30-to-40 out
 set internal idist 33
end-policy

route-policy 30-to-20 out
 set w
end-policy

route-policy 40-to-30 in
 set w
end-policy
```

Figure 7.6: gBGP configuration files for the network in Figure 7.2.

Router	Dest. prefix	Metric
10.10.0.1	10.10.0.1/32	<ecomm=0, epath=[], idist=0, ipath=[]>
	10.20.0.1/32	W
	10.30.0.1/32	W
	10.40.0.1/32	W
10.20.0.1	10.10.0.1/32	<ecomm=0, epath=[], idist=33, ipath=[10]>
	10.20.0.1/32	<ecomm=1, epath=[], idist=0, ipath=[]>
	10.30.0.1/32	W
	10.30.0.1/32	W
	10.40.0.1/32	W
10.30.0.1	10.10.0.1/32	<ecomm=0, epath=[1111], idist=0, ipath=[]>
	10.20.0.1/32	<ecomm=1, epath=[1111], idist=0, ipath=[]>
	10.30.0.1/32	<ecomm=0, epath=[], idist=0, ipath=[]>
	10.40.0.1/32	W
	10.40.0.1/32	W
10.40.0.1	10.10.0.1/32	<ecomm=0, epath=[1111], idist=77, ipath=[30]>
	10.20.0.1/32	<ecomm=1, epath=[1111], idist=77, ipath=[30]>
	10.30.0.1/32	<ecomm=0, epath=[], idist=77, ipath=[30]>
	10.40.0.1/32	<ecomm=0, epath=[], idist=0, ipath=[]>

Table 7.1: Routing tables produced by offline vector algorithm.

As well as linking the compiled C++ code with a gQuagga routing algorithm, we can use the same compiled library with a custom script written in Python that implements a simple offline non-distributed vector routing algorithm to help test and debug and examine the behaviour of a language. This tool can parse the configuration files from Figure 7.6, and output the routing tables at each step in the convergence process. Table 7.1 lists the final output for this example network, matching the routing tables produced by gBGP running the same configuration.

We will now extend this example to include route maps. First we update router 10.10.0.1 to originate some extra routes with destinations `10.10.0.2/32` and `10.10.0.3/32`. To implement the route map, we replace router `10.30.0.1`'s "`route-policy 20-to-30 in`" with the "`route-map`" expression listed in Figure 7.7. This aims to filter out routes to `10.10.0.3/32`, and to compute the import label's `idist` field based on the `ecomm` field of the wire metric received over the link.

The route map is an ERL expression that is dynamically compiled to C++ and then executed for each route received, with the output used in place of the import label. (Currently this is only implemented in the Python script, but the same approach could be applied in gQuagga.) In this example we write the ERL with full explicit types. `wir` and `lbl_i` are the standard types of the routing language based on our definition of the ERL code extracted from a RAML specification. A few shorthand types

Router	Dest. prefix	Metric
10.40.0.1	10.10.0.1/32	<ecomm=0, epath=[1111], idist=1077, ipath=[30]>
	10.10.0.2/32	<ecomm=0, epath=[1111], idist=1077, ipath=[30]>
	10.10.0.3/32	W
	10.20.0.1/32	<ecomm=1, epath=[1111], idist=2077, ipath=[30]>
	10.30.0.1/32	<ecomm=0, epath=[], idist=77, ipath=[30]>
	10.40.0.1/32	<ecomm=0, epath=[], idist=0, ipath=[]>

Table 7.2: Routing table for 10.40.0.1 in route-map example.

like wir_noninf were defined by hand elsewhere for lessened inconvenience; in this case wir = TyAddConst(W, wir_noninf), so the ExprConstCase expressions branch on whether the metric is a W or of type wir_noninf. This is evidently not a very usable language for writing route maps but it demonstrates the concepts.

Figure 7.2 lists the output at router 10.40.0.1 for this new network with the route map. The route to 10.10.0.3 is indeed filtered out (set to W), and the route to 10.20.0.1 (originated with ecomm=1) has a different idist as intended. The addition of route maps has therefore provided significant additional expressivity to our routing protocol's policy, with no changes to the specification of the routing language and without compromising the protocol's correctness properties determined by automatic verification of the algebraic language specification.

```
route-map 20-to-30 in
  fun (p:TyString m:wir) ->
    (* If the wire metric 'm' is W, return the label W *)
    ExprConstCase(m, W:wir, W:lbl_i,
      (* ... else process the non-W version of 'm' *)
      fun (m2:wir_noninf) ->

      (* If the destination prefix is in the given set, return the label W *)
      ExprCond(
        ExprUnop(UoPrefixIn("10.10.0.3/32"):TyBool, p):TyBool,
        W:lbl_i,

        (* Otherwise return a non-W external label *)
        ExprRest(
          ExprInject(external,
            ExprRecord(
              (* Metric's ecomm/epath were computed on export so they are empty here *)
              ecomm = ExprUnit : TyUnit,
              epath = ExprUnit : TyUnit,

              (* Set ipath to a non-W empty list *)
              ipath = ExprRest(ExprList() : TyListSimp(TyIntNonNeg)) : paths_t

              (* Compute idist based on the wire metric (as an arbitrary example) *)
              idist =
                (* Check if wire metric was internal or external *)
                ExprSwitch(m2,
                  internal = fun (m3:wir_int) -> 0 : TyIntNonNeg,
                  external = fun (m3:wir_ext) ->
                    (* If the wire metric's ecomm = 1 then return 2000, else 1000 *)
                    ExprCond(
                      ExprBinop(BoEq:TyBool,
                        ExprSelect(m3, ecomm):TyIntRange(0, 3), 1:TyIntRange(0, 3)
                      ) : TyBool,
                      2000 : TyIntNonNeg,
                      1000 : TyIntNonNeg
                    ) : TyIntNonNeg
                ) : TyIntNonNeg,

            ) : lbl_i_ext
          ) : lbl_i_noninf
        ) : lbl_i
      ) : lbl_i
    ) : lbl_i
end-map
```

Figure 7.7: Example route map written in ERL.

Chapter 8

Conclusions

8.1 Future work

The scope of the metarouting system allows a great many directions for improvements or extensions. This section will suggest some that are related to the work in this dissertation.

8.1.1 Extended language definitions

The version of RAML we defined in Section 3.2 is a small subset of what would be necessary in practice for many more varied routing languages. Billings [Bil09, Ch. 8] describes a larger version of RAML, and more recently Naudžiūnas [Nau11] details a version with extensive property inference implemented in Coq. These can largely be implemented with the version of ERL described in this dissertation, with the addition of a number of extra semigroup operators that follow the same patterns as the ones we defined here. A few other additions to ERL have been implemented and may be useful in some cases: enumeration types and associated operators (for example to replace the integer types in implementing the customer–provider–peer algebra from Section 2.2.6), product types (similar to records but without requiring explicit names, for convenience), and several unary operators. Integrating these into the definition of ERL is straightforward.

Further extensions to RAML were briefly discussed in Section 4.3 for modelling the behaviour of EIGRP (combining several metric components using addition to get a weight that can be compared), and Section 6.4 discussed the requirements for a user-friendly route map language that would be compatible with the metarouting system; these language design issues could benefit from further exploration.

8.1.2 Infinities

One detail we have glossed over in the scoped product example of Section 3.3.3 is the distinction between infinities (elements that are top in $\preceq$ and annihilators in $\triangleright$) at different levels of the algebra: a loop in the `epath` attribute can result in the metric $(0,\ \omega,\ 0,\ [])$ which is different from the top-level infinity metric ω. To handle this properly, it seems we need a mechanism to propagate infinities from sub-expressions out to the top level. One attempted solution is to define a new *absorbing* variant of lexicographic product

$$(S_1,\ L_1,\ \preceq_1,\ \triangleright_1) \vec{\times}_\omega (S_2,\ L_2,\ \preceq_2,\ \triangleright_2)$$

which produces a new algebra with metric type

$$S = \Big((S_1 - \{\omega_1\}) \times (S_2 - \{\omega_2\})\Big) \cup \{\omega\}$$

and a new $\triangleright$ which returns the metric ω if either $\triangleright_1$ or $\triangleright_2$ returns their corresponding ω. In practice the definition would need to be extended to cope with sub-algebras that do not have an infinity, as well as having some way to determine what the infinity value actually is. Currently the infinity value is identified only by the algebraic properties of the sub-expressions; this definition may result in the ERL semantics of a RAML expression having a dependency on the algebraic properties, which is not currently the case. It will also be necessary to derive inference rules for the algebraic properties of this new operator, as they differ from the standard lexicographic product.

It may be useful to let routing language specifications select the infinity-propagation behaviour on a field-by-field basis rather than choosing it for the entire product constructor. For example a language might be designed with a clamped distance field limited to the range $[0, 254] \cup \{\infty\}$ so it could be implemented efficiently in a single byte, where ∞ represents a route that is unusually long and no longer able to participate usefully in shortest-path-length decisions but is still perfectly acceptable for routing; it would be necessary to specify that this infinity does not propagate out to a global infinity, while other fields (such as paths) in the same lexicographic product may need to.

The difficulty with introducing the absorbing lexicographic product into metarouting is that we will require if-and-only-if property inference rules for it, which are non-trivial. Parsonage et al. [PNR11] suggest rules but only for absorbing infinities from the first component, not for the more difficult and more important general case. As an alternative, Gurney [GG11] introduces a method for handling errors through *reductions*: given a routing algebra S, an arbitrary subset E of metrics (for example those containing infinities in certain product fields) can be mapped onto a single infinity value, as part of a new algebra $\mathbf{err}(S,\ E)$. We could implement equivalent behaviour in our

current metarouting system using the route maps of Chapter 6, wrapping $f(d,\ m)$ into a new $f_E(d,\ m)$ that first tests whether $f(d,\ m) \rhd m \in E$ and returns the infinity label in that case, else continues with the standard processing, but a more elegant and automatic solution is desirable.

8.1.3 ERL algebraic properties

Billings [Bil09, Ch. 8] begins to explore some compiler optimisations that are possible when we know the algebraic properties of the ERL code being compiled. In particular, given an order transform $(S,\ L,\ \preceq,\ \rhd)$, a minimal set $M \in \wp_{\preceq}(S)$, and a label $l \in L$, a useful operation is to map the transform over the set giving $\{l \rhd s \mid s \in M\}$. In general it will be necessary to re-minimise the output to ensure it is still a minimal set, but if the order transform has the distributivity property then there is no need to re-minimise. Knowing the properties therefore allows the ERL compiler to omit unnecessary code and make better optimisation decisions.

Our updated definition of languages in Chapter 3 associates algebraic properties only with RAML expressions, not with ERL expressions (which may not even represent structures such as ordered transforms in which properties are meaningful), and so the ERL compiler cannot be responsible for this optimisation by itself. One option would be to reintroduce the concept of properties into ERL. As we should perform these optimisations even when the relevant expressions are nested inside a more complex routing language, we would need a way to refer to subexpressions in the ERL types and preorders and transforms and associate them with the relevant properties (computed by the RAML-to-ERL compiler), then have the ERL compiler extract these properties when it is generating code.

Alternatively we could make this the responsibility of the RAML-to-ERL compiler: instead of translating a `minset_union_map` RAML expression into a `UoSetMap` ERL operator, it would first check the algebraic properties of the RAML sub-expression and translate into a new `UoSetMapDistributive` operator to select the optimised implementation when it knows it to be safe. This would better preserve the distinction between RAML and ERL, in that only RAML knows about algebraic properties and ERL merely represents the computational content of the algebra, but it would also force the RAML compiler to understand these low-level optimisations.

8.1.4 Multipath routing

In Chapter 4 we said that the algorithms require an algebra with a total order, so that there is always a single best route. If we relax this requirement then we can get a model

of multipath routing: there may be multiple routes that are equivalent or incomparable, and the router could use any or all of them for forwarding traffic. Alternatively we could implement the routing algebras as semigroup transforms $(S, \oplus, \triangleright)$: instead of a partial order $\preceq$ over metrics S, use a (non-selective) semigroup operator $\cup$ over minimal sets $\wp_{\preceq}(S)$. This is more powerful than the equal-cost multipath (ECMP) provided by some implementations of protocols such as OSPF and BGP – a larger range of paths can be made available for forwarding since we do not require equality – but in most cases it would require a forwarding mechanism based on tunnels rather than the basic next-hop forwarding model we have assumed in this dissertation, to ensure the forwarding paths exactly match the selected routes.

In either case we would need to extend the routing algorithms to support multipath routing. There has been some recent work in the IETF to add a multipath mechanism to BGP [WCRS11, VdSFB10], which could be reused for a multipath gBGP based on partial orders. Multipath based on min-sets semigroups could provide more power, including a kind of aggregation of multiple min-set entries into a single entry to avoid a potentially exponential expansion in set sizes, but would require a much more complex interaction between the routing language implementation and the forwarding plane of the algorithm implementation.

8.2 Summary

The metarouting project began with the idea of implementing routing protocols using declarative specifications based on algebraic routing theory, balancing carefully between expressivity and the tractability of providing correctness guarantees. Over time, the scope and complexity of this concept have become apparent and have grown significantly in terms of both the theory and the practical implementation, but the core idea has been strong enough to evolve and extend to support this. In this dissertation we detailed some components of the implementation (the language semantics in Chapter 3 and the generalised Quagga algorithms in Chapter 4) and many of the design decisions they involved, building up a working prototype implementation (demonstrated in Chapter 7), which provides many development challenges in itself and helps to solidify the theory by highlighting any unresolved gaps. Focusing on vector routing protocols, which can efficiently implement complex policies due to their use of the distributed Bellman-Ford algorithm, we then extended the theory and implementation to develop features that are important in current protocol designs, namely the handling of complex policy that is split across export/import interfaces and that can be implemented using route maps (Chapters 5 and 6). This provides both a framework for understanding the capabilities and limits of protocol designs, and an implementation for rapidly experimenting with protocols based on potentially radically new routing

languages.

Although there are still many aspects of routing as practised today or as proposed for future networks that have not yet been incorporated into the metarouting model and that may introduce significant challenges of their own, the experience of developing a practical implementation of the model and extending it to support new features for vector protocols has shown this to be a viable approach and likely a fruitful direction for future research.

Bibliography

[AGLAB94] Bob Albrightson, J.J. Garcia-Luna-Aceves, and Joanne Boyle. EIGRP - a fast routing protocol based on distance vectors. In *Proc. Networld/Interop 94*, 1994.

[Ali11] M. Abdul Alim. *On the Interaction of Internet Routing Protocols*. PhD thesis, University of Cambridge, 2011.

[AVG+99] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing Policy Specification Language (RPSL). RFC 2622 (Proposed Standard), June 1999. Updated by RFC 4012.

[BC04] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc, 2004.

[BCC06] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456 (Draft Standard), April 2006.

[Bel58] Richard E. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BFM+05] H. Böhm, A. Feldmann, O. Maennel, C. Reiser, R. Volk, and D. Telekom. Network-wide inter-domain routing policies: Design and realization. In *Proc. NANOG*, volume 34, 2005.

[BFMR10] K. Butler, T. R. Farley, P. McDaniel, and J. Rexford. A Survey of BGP Security Issues and Solutions. *Proceedings of the IEEE*, 98(1):100–122, January 2010.

[BG92] Dimitri Bertsekas and Robert Gallager. *Data networks (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[BG03] Randy Bush and Timothy G. Griffin. Integrity for virtual private routed networks. In *In Proc. IEEE INFOCOM*, 2003.

[BG09] John N. Billings and Timothy G. Griffin. A model of internet routing using semi-modules. In *Proceedings of the 11th International Conference on Relational Methods in Computer Science and 6th International Conference on Applications of Kleene Algebra: Relations and Kleene Algebra in Computer Science*, RelMiCS '09/AKA '09, pages 29–43, Berlin, Heidelberg, 2009. Springer-Verlag.

[BH06] Andrea Bittau and Mark Handley. Decoupling policy from protocols. `http://www.xorp.org/papers/policy.pdf`, 2006.

[Bil09] J. Billings. *Specifying and compiling Internet routing protocols*. PhD thesis, University of Cambridge, 2009.

[bir] The BIRD Internet Routing Daemon. `http://bird.network.cz/`.

[BMH11] Uli Bornhauser, Peter Martini, and Martin Horneffer. The Scope of the IBGP Routing Anomaly Problem. In Norbert Luttenberger and Hagen Peters, editors, *17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011)*, volume 17 of *OpenAccess Series in Informatics (OASIcs)*, pages 2–13, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BWM08] Vijay Bollapragada, Russ White, and Curtis Murphy. *Inside Cisco IOS Software Architecture*. Cisco Press, 1 edition, 2008.

[Car79] Bernard Carré. *Graphs and Networks*. Oxford University Press, 1979.

[Chr11] J. Chroboczek. The Babel Routing Protocol. RFC 6126 (Experimental), April 2011.

[cis] Cisco IOS. `http://www.cisco.com/`.

[Cis04] Cisco. *Enhanced Interior Gateway Routing Protocol*, 2004. `http://www.cisco.com/en/US/tech/tk365/technologies_white_paper09186a0080094cb7.shtml`.

[Cis06] Cisco. *Cisco Internetworking Operating Systems (IOS)*, 2006. `http://www.cisco.com/en/US/products/sw/iosswrel/ps1818/products_tech_note09186a0080135951.shtml`.

[Cis09a] Cisco. *Cisco IOS IP Command Reference – bgp dampening*, 2009. `http://www.cisco.com/en/US/docs/ios/12_3/iproute/command/reference/ip2_a1g.html#wp1039828`.

[Cis09b] Cisco. *Cisco IOS IP Command Reference – bgp deterministic-med*, 2009. `http://www.cisco.com/en/US/docs/ios/12_3/iproute/command/reference/ip2_a1g.html#wp1040132`.

[Cis11] Cisco. *Cisco IOS IP Routing: Protocol-Independent Command Reference – set metric*, 2011. `http://www.cisco.com/en/US/docs/ios/iproute_pi/command/reference/iri_pi2.html#wp1013228`.

[Cla88] D. Clark. The design philosophy of the darpa internet protocols. In *Symposium proceedings on Communications architectures and protocols*, SIGCOMM '88, pages 106–114, New York, NY, USA, 1988. ACM.

[CLR92] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms (First Edition)*. The MIT Press, 1992.

[CR08] E. Chen and Y. Rekhter. Outbound Route Filtering Capability for BGP-4. RFC 5291 (Proposed Standard), August 2008.

[CS08] E. Chen and S. Sangli. Address-Prefix-Based Outbound Route Filter for BGP-4. RFC 5292 (Proposed Standard), August 2008.

[CTL96] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. RFC 1997 (Proposed Standard), August 1996.

[DB08] Benoit Donnet and Olivier Bonaventure. On BGP communities. *SIGCOMM Comput. Commun. Rev.*, 38(2):55–59, 2008.

[DD10] E. Davies and A. Doria. Analysis of Inter-Domain Routing Requirements and History. RFC 5773 (Historic), February 2010.

[Del11] Dell'Oro Group. Service provider router market rebounded with 21 percent growth in 2010, 2011. `http://www.delloro.com/news/2011/Rtr022811.htm`.

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1*, pages 269–271, 1959.

[EMG+07] William Enck, Patrick McDaniel, Albert Greenberg, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, and Sanjay Rao. Configuration management at massive scale: System design and experience. In *USENIX ATC*, pages 73–86, 2007.

[FB05] Nick Feamster and Hari Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[FF62] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[FL06] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice), August 2006.

[Gao00] Lixin Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking*, 9:733–745, 2000.

[GG07] A. Gurney and T. G. Griffin. Lexicographic products in metarouting. In *ICNP*, October 2007.

[GG08] T. G. Griffin and A. Gurney. Increasing bisemigroups and algebraic routing. In *10th International Conference on Relational Methods in Computer Science (RelMiCS10)*, April 2008.

[GG11] Alexander J. T. Gurney and Timothy G. Griffin. Pathfinding through congruences. In *Proceedings of the 12th international conference on Relational and algebraic methods in computer science*, RAMICS'11, pages 180–195, Berlin, Heidelberg, 2011. Springer-Verlag.

[GH05] T. Griffin and G. Huston. BGP Wedgies. RFC 4264 (Informational), November 2005.

[GLA93] J.J. Garcia-Luna-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1(1), 1993.

[GM84] M. Gondran and M. Minoux. *Graphs and Algorithms*. Wiley, 1984.

[GM08] M. Gondran and M. Minoux. *Graphs, Dioids, and Semirings. New Models and Algorithms*. Springer, 2008.

[GR01] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on Networking*, pages 681–692, December 2001.

[Gri10] T. G. Griffin. The stratified shortest-paths problem. In *The third International Conference on COMmunication Systems and NETworkS (COMSNETS)*, January 2010.

[GS02] Barry Raveendran Greene and Philip Smith. *Cisco ISP Essentials*. Cisco Press, 2002.

[GS03] Mohamed G. Gouda and Marco Schneider. Maximizable routing metrics. *IEEE/ACM Transactions on Networking*, 11(4):663–675, August 2003.

[GS05] Timothy G. Griffin and João Luís Sobrinho. Metarouting. In *Proc. ACM SIGCOMM*, August 2005.

[GSW99] T. G. Griffin, F. B. Shepherd, and G. Wilfong. Policy disputes in path-vector protocols. In *Proc. Inter. Conf. on Network Protocols*, November 1999.

[GSW02] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking*, 10(2):232–243, April 2002.

[Gur09] Alexander J. T. Gurney. *Construction and verification of routing algebras*. PhD thesis, University of Cambridge, 2009.

[GW99] T. G. Griffin and G. Wilfong. An analysis of BGP convergence properties. In *Proc. ACM SIGCOMM*, September 1999.

[GW02] T. G. Griffin and G. Wilfong. An analysis of the MED oscillation problem in BGP. In *Proc. Inter. Conf. on Network Protocols*, 2002.

[Hen88] C. Hendrick. Routing information protocol. RFC 1058, 1988.

[HKG+05] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible IP routing software. In *Networked Systems Design and Implementation (NSDI)*, May 2005.

[Hus11] Geoff Huston. Bgp routing table analysis reports, 2011. `http://bgp.potaroo.net/`.

[ian11] Border Gateway Protocol (BGP) Parameters, 2011. `http://www.iana.org/assignments/bgp-parameters`.

[IEE01] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6.

[iso02] ISO/IEC 10589 (Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)), 2002.

[jun] Juniper Junos. `http://www.juniper.net/products_and_services/junos/`.

[Jun10a] Juniper. *JUNOS 10.0 Policy Framework Configuration Guide*, 2010. `http://www.juniper.net/techpubs/en_US/junos10.0/`

information-products/topic-collections/config-guide-policy/policy-configuring-actions-in-routing-policy-terms.html#jd0e6901.

[Jun10b] Juniper. *JUNOS 10.0 Routing Protocols Configuration Guide*, 2010. http://www.juniper.net/techpubs/en_US/junos10.0/information-products/topic-collections/config-guide-routing/routing-configuring-routing-table-path-selection-for-bgp.html.

[Ker08] Sean Michael Kerner. JUNOS: Open, but Not Open Source. *InternetNews*, 2008. http://www.internetnews.com/dev-news/article.php/3759741.

[LHSR05] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM*, 2005.

[LR89] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). RFC 1105 (Experimental), June 1989. Obsoleted by RFC 1163.

[LXP+08] Franck Le, Geoffrey G. Xie, Dan Pei, Jia Wang, and Hui Zhang. Shedding light on the glue logic of the internet routing architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 39–50, New York, NY, USA, 2008. ACM.

[LXZ07] Franck Le, Geoffrey Xie, and Hui Zhang. Understanding route redistribution. In *Proc. Inter. Conf. on Network Protocols*, 2007.

[LXZ10] Franck Le, Geoffrey G. Xie, and Hui Zhang. Theory and new primitives for safely connecting routing protocol instances. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 219–230, New York, NY, USA, 2010. ACM.

[Mal98] G. Malkin. RIP Version 2. RFC 2453 (Standard), November 1998. Updated by RFC 4822.

[MG06] D. McPherson and V. Gill. BGP MULTI_EXIT_DISC (MED) Considerations. RFC 4451 (Informational), March 2006.

[MGWR02] D. McPherson, V. Gill, D. Walton, and A. Retana. Border Gateway Protocol (BGP) Persistent Route Oscillation Condition. RFC 3345 (Informational), August 2002.

[Moy98] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998. Updated by RFC 5709.

[MSO+99] D. Meyer, J. Schmitz, C. Orange, M. Prior, and C. Alaettinoglu. Using RPSL in Practice. RFC 2650 (Informational), August 1999.

[Nau11] Vilius Naudžiūnas. Design and implementation of a language for path algebras. PhD dissertation, University of Cambridge, forthcoming, 2011.

[NG11] Vilius Naudžiūnas and Timothy G. Griffin. An implementation of metarouting using coq. In *WRiPE 2011: 1st International Workshop on Rigorous Protocol Engineering, co-located with ICNP 2011*, 2011.

[NJW+10] Vivek Nigam, Limin Jia, Anduo Wang, Boon Thau Loo, and Andre Scedrov. An operational semantics for network datalog. Technical report, University of Pennsylvania Department of Computer and Information Science, 2010.

[PBRD03] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[PNR11] Eric Parsonage, Hung X. Nguyen, and Matthew Roughan. Absorbing lexicographic products in metarouting. In *WRiPE 2011: 1st International Workshop on Rigorous Protocol Engineering, co-located with ICNP 2011*, 2011.

[qua] Quagga routing suite. `http://www.quagga.net/`.

[RL94] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1654 (Proposed Standard), July 1994. Obsoleted by RFC 1771.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006.

[RR06] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks (VPNs). RFC 4364 (Proposed Standard), February 2006. Updated by RFCs 4577, 4684, 5462.

[SCE+05] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Morley Mao, Scott Shenker, and Ion Stoica. Hlp: a next generation inter-domain routing protocol. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 13–24, New York, NY, USA, 2005. ACM.

[Sob02] Joao Luis Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop. *IEEE/ACM Transactions on Networking*, 10(4):541–550, August 2002.

[Sob03] Joao Luis Sobrinho. Network routing with path vector protocols: Theory and applications. In *Proc. ACM SIGCOMM*, September 2003.

[Sob05] Joao Luis Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13(5):1160–1173, October 2005.

[TG09] Philip J. Taylor and Timothy G. Griffin. A model of configuration languages for routing protocols. In *PRESTO '09: Proceedings of the 2nd ACM SIGCOMM workshop on Programmable routers for extensible services of tomorrow*, pages 55–60, New York, NY, USA, 2009. ACM.

[VC07] Q. Vohra and E. Chen. BGP Support for Four-octet AS Number Space. RFC 4893 (Proposed Standard), May 2007.

[VCG98] C. Villamizar, R. Chandra, and R. Govindan. BGP Route Flap Damping. RFC 2439 (Proposed Standard), November 1998.

[VdSFB10] Virginie Van den Schrieck, Pierre Francois, and Olivier Bonaventure. Bgp add-paths: the scaling/performance tradeoffs. *IEEE J.Sel. A. Commun.*, 28:1299–1307, October 2010.

[VGE00] Kannan Varadhan, Ramesh Govindan, and Deborah Estrin. Persistent route oscillations in inter-domain routing. *Computer Networks*, 32:1–16, 2000.

[VH09] Andreas Voellmy and Paul Hudak. Nettle: A language for configuring routing networks. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 211–235, Berlin, Heidelberg, 2009. Springer-Verlag.

[WCRS11] D. Walton, E. Chen, A. Retana, and J. Scudder. Advertisement of Multiple Paths in BGP, September 2011. `http://tools.ietf.org/html/draft-ietf-idr-add-paths-06`.

[Whe03] Jeff S. Wheeler. MED intent, purpose, and modern usage. quagga-users mailing list, 2003. `http://lists.quagga.net/pipermail/quagga-users/2003-November/000896.html`.

[WJA04] Tina Wong, Van Jacobson, and Cengiz Alaettinoglu. Making sense of BGP. In *In Nanog presentation*, 2004.

[WMS05] Russ White, Danny McPherson, and Srihari Sangli. *Practical BGP*. Addison Wesley, 2005.

[XG04] Jianhong Xia and Lixin Gao. On the evaluation of as relationship inferences. In *In Proc. of IEEE GLOBECOM*, pages 1373–1377, 2004.

[Zmi09] Earl Zmijewski. AfNOG takes byte out of internet, 2009. `http://www.renesys.com/blog/2009/05/byte-me.shtml`.

Appendix A

Full ERL definition

A.1 Types

Type expression t	Algebraic semantics $[\![t]\!]$
`TyUnit`	$\mathbf{1}$
`TyBool`	$\{\top, \bot\}$
`TyInt`	$\mathbb{Z}$
`TyIntNonNeg`	$\mathbb{N}$
`TyIntPos`	$\mathbb{N}^+$
`TyIntRange(`n`, `m`)`	$\{i \in \mathbb{Z} \mid n \leq i \leq m\}$
`TyString`	set of all opaque character strings
`TyList(`t'`)`	$\{[s_1, \ldots, s_n] \mid n \in \mathbb{N} \wedge s_i \in [\![t']\!]\}$

Type expression t	Algebraic semantics $[\![t]\!]$
`TyListSimp(`t'`)`	$\{[s_1, \ldots, s_n] \mid n \in \mathbb{N} \wedge s_i \in [\![t']\!] \wedge s_i = s_j \Rightarrow i = j\}$
`TySet(`t'`)`	$\wp([\![t']\!])$
`TySetMin(`t'`, `p`)`	$\wp_{[\![p]\!]}([\![t']\!])$
`TyAddConst(`c`, `t'`)`	$\{c\} \cup [\![t']\!]$
`TyRecord(`n_1`=`t_1`, `n_2`=`t_2`)`	$[\![t_1]\!] \times [\![t_2]\!]$
`TyUnion(`n_1`=`t_1`, `n_2`=`t_2`)`	$[\![t_1]\!] \uplus [\![t_2]\!]$

A.2 Preorders

Preorder expression p	Algebraic semantics $[\![p]\!]$
`PoIntLte : `t	$\leq$
`PoListLenLte : `t	$\preceq$ where $s_1 \preceq s_2 \Leftrightarrow \lvert s_1 \rvert \leq \lvert s_2 \rvert$
`PoAddTop(`p'`:`t`) : TyAddConst(`c`, `t`)`	$\preceq$ where $s_1 \preceq s_2 \Leftrightarrow s_2 = c \ \vee \ (s_1 \neq c \wedge s_2 \neq c \wedge s_1 \, [\![p']\!] \, s_2)$
`PoAddBot(`p'`:`t`) : TyAddConst(`c`, `t`)`	$\preceq$ where $s_1 \preceq s_2 \Leftrightarrow s_1 = c \ \vee \ (s_1 \neq c \wedge s_2 \neq c \wedge s_1 \, [\![p']\!] \, s_2)$
`PoDual(`p'`:`t`) : `t	$\preceq$ where $s_1 \preceq s_2 \Leftrightarrow s_2 \, [\![p']\!] \, s_1$
`PoEquiv : `t	$\preceq$ where $s_1 \preceq s_2$ for all s_1, s_2
`PoIncomp : `t	$\preceq$ where $s_1 \preceq s_2 \Leftrightarrow s_1 = s_2$
`PoRecord(`n_1`=`p_1`:`t_1`, `n_2`=`p_2`:`t_2`) : TyRecord(`n_1`=`t_1`, `n_2`=`t_2`)`	$\preceq$ where $(u_1, u_2) \preceq (v_1, v_2) \Leftrightarrow u_1 [\![p_1]\!] v_1 \ \wedge \ u_2 [\![p_2]\!] v_2$

Preorder expression p	Algebraic semantics $[\![p]\!]$
`PoRecordLex(`n_1`=`p_1`:`t_1`, `n_2`=`p_2`:`t_2`)` `  : TyRecord(`n_1`=`t_1`, `n_2`=`t_2`)`	$\preceq$ where $(u_1, u_2) \preceq (v_1, v_2) \Leftrightarrow u_1 \prec_1 v_1 \vee (u_1 \preceq_1 v_1 \wedge u_2 \preceq_2 v_2)$ with $\preceq_1 \equiv [\![p_1]\!]$ and $\preceq_2 \equiv [\![p_2]\!]$
`PoUnion(`n_1`=`p_1`:`t_1`, `n_2`=`p_2`:`t_2`)` `  : TyUnion(`n_1`=`t_1`, `n_2`=`t_2`)`	$\preceq$ where $\text{inl}(u_1) \preceq \text{inl}(v_1) \Leftrightarrow u_1 [\![p_1]\!] v_1$ $\text{inr}(u_2) \preceq \text{inr}(v_2) \Leftrightarrow u_2 [\![p_2]\!] v_2$
`PoUnionOrdered(`n_1`=`p_1`:`t_1`, `n_2`=`p_2`:`t_2`)` `  : TyUnion(`n_1`=`t_1`, `n_2`=`t_2`)`	$\preceq$ where $\text{inl}(u_1) \preceq \text{inl}(v_1) \Leftrightarrow u_1 [\![p_1]\!] v_1$ $\text{inr}(u_2) \preceq \text{inr}(v_2) \Leftrightarrow u_2 [\![p_2]\!] v_2$ $\text{inl}(u_1) \preceq \text{inr}(v_2)$

A.3 Semigroups

Semigroup expression g	Algebraic semantics $[\![g]\!]$
`SgIntPlus : `t	$+$
`SgIntRangePlus` `  : TyAddConst(`c`, TyIntRange(`n`, `m`))`	$\otimes$ where $c \otimes i = c$ $i \otimes c = c$ $i_1 \otimes i_2 = i_1 + i_2$ if $n \leq i_1 + i_2 \leq m$ $i_1 \otimes i_2 = c$ otherwise
`SgIntMin : `t	min
`SgIntMax : `t	max

Semigroup expression g	Algebraic semantics $[\![g]\!]$
`SgAddAlpha(`g'`:`t`) : TyAddConst(`c`, `t`)`	$\otimes$ where for all $s, s_1, s_2 \in [\![t]\!]$, $c \otimes s = s$ $s \otimes c = s$ $s_1 \otimes s_2 = s_1\ [\![g']\!]\ s_2$
`SgAddOmega(`g'`:`t`) : TyAddConst(`c`, `t`)`	$\otimes$ where for all $s, s_1, s_2 \in [\![t]\!]$, $c \otimes s = c$ $s \otimes c = c$ $s_1 \otimes s_2 = s_1\ [\![g']\!]\ s_2$
`SgLeft : `t	$\otimes$ where for all $s_1, s_2 \in [\![t]\!]$, $s_1 \otimes s_2 = s_1$
`SgRight : `t	$\otimes$ where for all $s_1, s_2 \in [\![t]\!]$, $s_1 \otimes s_2 = s_2$

A.4 Transforms

Transform expression f	Algebraic semantics $[\![f]\!]$
`fun (`a_1`:`t_1 a_2`:`t_2 `...) -> `f	$\lambda a_1 a_2.\ [\![f]\!]$
ident	the value of *ident* in the current variable binding
`ExprUnit : TyUnit`	1
`ExprInt(`n`) : `t	n
`ExprString("`s`") : `t	s
`ExprList(`f_1`, `f_2`, ...) : `t	$[[\![f_1]\!], [\![f_2]\!], \ldots]$
`ExprSet(`f_1`, `f_2`, ...) : `t	$\{[\![f_1]\!], [\![f_2]\!], \ldots\}$

Transform expression f	Algebraic semantics $[\![f]\!]$
`ExprBinop(BoListCons,` `  `f_1`:t, `f_2`:TyList(t)` `) : TyList(t)`	$[\![f_1]\!] :: [\![f_2]\!]$
`ExprBinop(BoListCons,` `  `f_1`:t, `f_2`:TyAddConst(c, TyListSimp(t))` `) : TyAddConst(c, TyListSimp(t))`	$\begin{cases} c & \text{if } [\![f_1]\!] \text{ is in list } [\![f_2]\!] \\ [\![f_1]\!] :: [\![f_2]\!] & \text{otherwise} \end{cases}$
`ExprBinop(BoSemigroup(`g`), `f_1`, `f_2`) : t`	$[\![f_1]\!]\ [\![g]\!]\ [\![f_2]\!]$
`ExprApply(`f'`:t, `a_1`, ...) : t`	$[\![f']\!](a_1, \ldots)$
`ExprCond(`f'`:TyBool, `f_1`, `f_2`) : t`	$\begin{cases} [\![f_1]\!] & \text{if } [\![f']\!] = \top \\ [\![f_2]\!] & \text{if } [\![f']\!] = \bot \end{cases}$
`ExprRecord(`n_1`=`f_1`, `n_2`=`f_2`) : t`	$([\![f_1]\!], [\![f_2]\!])$
`ExprSelect(`f'`:TyRecord(`n_1`=`t_1`, `n_2`=`t_2`), `n`)`	$\begin{cases} f_1 & \text{if } n = n_1 \\ f_2 & \text{if } n = n_2 \end{cases}$ where $[\![f']\!] = (f_1, f_2)$
`ExprConstCase(`f'`:TyAddConst(`c`, `t`), `c`, `f_1`, `f_2`)`	$\begin{cases} [\![f_1]\!] & \text{if } [\![f']\!] = c \\ [\![f_2]\!]([\![f']\!]) & \text{otherwise} \end{cases}$
`ExprInject(`n`, `f'`) : TyUnion(`n_1`=`t_1`, `n_2`=`t_2`)`	$\begin{cases} \text{inl}([\![f']\!]) & \text{if } n = n_1 \\ \text{inr}([\![f']\!]) & \text{if } n = n_2 \end{cases}$
`ExprSwitch(`f'`:TyUnion(`n_1`=`t_1`, `n_2`=`t_2`),` `  `n_1`=`f_1`, `n_2`=`f_2`) : t`	$\begin{cases} [\![f_1]\!](g) & \text{if } [\![f']\!] = \text{inl}(g) \\ [\![f_2]\!](g) & \text{if } [\![f']\!] = \text{inr}(g) \end{cases}$

Appendix B

Full RAML definition

RAML expression e	ERL translation $(\!\lvert e \rvert\!)$
`int_min_plus(`t`)`	`sig=TyAddConst(W, `t`)` `lbl=TyAddConst(W, `t`)` `ord=PoAddTop(PoIntLte)` `tfm=fun (l s) -> ExprBinop(BoSemigroup(SgAddOmega(SgIntPlus)), l, s)`
`int_range_min_plus(`n`, `m`)`	`sig=TyAddConst(W, TyIntRange(`n`, `m`))` `lbl=TyAddConst(W, TyIntRange(`n`, `m`))` `ord=PoAddTop(PoIntLte)` `tfm=fun (l s) -> ExprBinop(BoSemigroup(SgIntRangePlus), l, s)`

RAML expression e	ERL translation $(\!\|e\|\!)$
`int_max_min(`t`)`	`sig=TyAddConst(W, `t`)` `lbl=TyAddConst(W, `t`)` `ord=PoAddTop(PoDual(PoIntLte))` `tfm=fun (l s) -> ExprBinop(BoSemigroup(SgAddOmega(SgIntMin)), l, s)`
`paths(`t`)`	`sig=TyAddConst(NOTSIMPLE, TyListSimp(`t`))` `lbl=`t `ord=PoAddTop(PoListLenLte)` `tfm=fun (l s) -> ExprBinop(BoListCons, l, s)`
`add_top(`c`, `e'`)`	`sig=TyAddConst(`c`, `$(\!\|e'\|\!)_{\texttt{sig}}$`)` `lbl=TyAddConst(`$c$`, `$(\!\|e'\|\!)_{\texttt{lbl}}$`)` `ord=PoAddTop(`$(\!\|e'\|\!)_{\texttt{ord}}$`)` `tfm=fun (l s) ->` `  ExprConstCase(l, `$c$`, `$c$`, fun (l2) ->` `  ExprConstCase(s, `$c$`, `$c$`, fun (s2) ->` `  ExprApply(`$(\!\|e'\|\!)_{\texttt{tfm}}$`, l2, s2)))`
`right(`e'`)`	`sig=`$(\!\|e'\|\!)_{\texttt{sig}}$ `lbl=TyUnit` `ord=`$(\!\|e'\|\!)_{\texttt{ord}}$ `tfm=fun (l s) -> s`

RAML expression e	ERL translation $(\!\|e\|\!)$
`left(`e'`)`	`sig=`$(\!\|e'\|\!)_{\texttt{sig}}$ `lbl=`$(\!\|e'\|\!)_{\texttt{sig}}$ `ord=`$(\!\|e'\|\!)_{\texttt{ord}}$ `tfm=fun (l s) -> l`
`lex_product(`n_1`=`e_1`, `n_2`=`e_2`)`	`sig=TyRecord(`n_1`=`$(\!\|e_1\|\!)_{\texttt{sig}}$`, `$n_2$`=`$(\!\|e_2\|\!)_{\texttt{sig}}$`)` `lbl=TyRecord(`$n_1$`=`$(\!\|e_1\|\!)_{\texttt{lbl}}$`, `$n_2$`=`$(\!\|e_2\|\!)_{\texttt{lbl}}$`)` `ord=PoRecordLex(`$n_1$`=`$(\!\|e_1\|\!)_{\texttt{ord}}$`, `$n_2$`=`$(\!\|e_2\|\!)_{\texttt{ord}}$`)` `tfm=fun (l s) ->` `  ExprRecord(` `    `$n_1$`=ExprApply(`$(\!\|e_1\|\!)_{\texttt{tfm}}$`,` `       ExprSelect(l, `$n_1$`), ExprSelect(s, `$n_1$`)),` `    `$n_2$`=ExprApply(`$(\!\|e_2\|\!)_{\texttt{tfm}}$`,` `       ExprSelect(l, `$n_2$`), ExprSelect(s, `$n_2$`))` `  )`
`function_union(`n_1`=`e_1`, `n_2`=`e_2`)`	`sig=`$(\!\|e_1\|\!)_{\texttt{sig}}$ (requires $(\!\|e_1\|\!)_{\texttt{sig}} = (\!\|e_2\|\!)_{\texttt{sig}}$) `lbl=TyUnion(`$n_1$`=`$(\!\|e_1\|\!)_{\texttt{lbl}}$`, `$n_2$`=`$(\!\|e_2\|\!)_{\texttt{lbl}}$`)` `ord=PoUnion(`$n_1$`=`$(\!\|e_1\|\!)_{\texttt{ord}}$`, `$n_2$`=`$(\!\|e_2\|\!)_{\texttt{ord}}$`)` `tfm=fun (l s) ->` `  ExprSwitch(l,` `    `$n_1$` = fun (l1) -> ExprApply(`$(\!\|e_1\|\!)_{\texttt{tfm}}$`, l1, s),` `    `$n_2$` = fun (l2) -> ExprApply(`$(\!\|e_2\|\!)_{\texttt{tfm}}$`, l2, s)` `  )`

Appendix C

Route map command syntaxes

```
area area-id;
as-path [ regular-expression-names ];
as-path-group [ as-path-group-names ];
color preference;
color2 preference;
community [ community-names ];
external type (1 | 2);
family family-name;
instance instance-name;
interface [ interface-names ];
level isis-level;
local-preference value;
metric metric-value;
metric2 metric-value;
metric3 metric-value;
metric4 metric-value;
neighbor [ ip-addresses ];
next-hop [ ip-addresses ];
origin (egp | igp | incomplete);
policy [ policy-names ];
preference preference;
preference2 preference;
protocol [ protocol-names ];
rib routing-table-name;
tag [ tag-numbers ];
tag2 tag-number;
```

Figure C.1: JunOS 10.1 policy match conditions.

```
(accept | reject);
as-path-expand (as-number | last-as) <count number>;
as-path-prepend as-number;
class class-name;
color (preference | add number | subtract number);
color2 (preference | add number | subtract number);
community (add | delete | set | + | - | =) community-name;
cos-next-hop-map map-name;
damping list-name;
default-action (accept | reject);
destination-class class-name;
external type (1 | 2);
forwarding-class class-name;
install-nexthop <strict> (lsp [ lsp-names ] | lsp-regex [ regular-expressions ])
    <except (lsp [ lsp-names ] | lsp-regex [ regular-expressions ])>;
load-balance per-packet;
local-preference (preference | add number | subtract number);
metric (metric-value | add number | | subtract number
    | igp <metric-offset> | minimum-igp <metric-offset>
    | expression {
        metric (multiplier number | offset number | multiplier number offset number);
        metric2 (multiplier number | offset number | multiplier number offset number);
      }
metric2 (metric-value | add number | subtract number);
metric3 (metric-value | add number | subtract number);
metric4 (metric-value | add number | subtract number);
next (policy | term);
next-hop (ip-address | discard | next-table routing-table-name
         | peer-address | reject | self);
origin (egp | igp | incomplete);
preference (preference | add number | subtract number);
preference2 (preference | add number | subtract number);
priority (high | low | medium);
source-class class-name;
tag (tag-number | add number | subtract number);
tag2 (tag-number | add number | subtract number);
trace;
```

Figure C.2: JunOS 10.1 policy terms.

```
policy {
  policy-statement @: txt {
    term @: txt {
      from {
        as-path:              txt;
        as-path-list:         txt;
        community:            txt;
        community-list:       txt;
        neighbor:             ipv4range;
        origin:               u32;
        med:                  u32range;
        localpref:            u32range;
      }
      to {
        as-path:              txt;
        as-path-list:         txt;
        community:            txt;
        neighbor:             ipv4range;
        origin:               u32;
        med:                  u32range;
        localpref:            u32range;
        was-aggregated:       bool;
      }
      then {
        as-path-prepend:      u32;
        as-path-expand:       u32;
        community:            txt;
        community-add:        txt;
        community-del:        txt;
        origin:               u32;
        med:                  u32;
        med-remove:           bool;
        localpref:            u32;
        aggregate-prefix-len: u32;
        aggregate-brief-mode: bool;
      }
    }
  }
}
```

Figure C.3: XORP BGP policy terms.