Specifying and compiling Internet routing protocols

John N. Billings

University of Cambridge
Computer Laboratory
Queens' College

October 2009

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.

Publications

John N. Billings, Timothy G. Griffin. A model of Internet routing using semi-modules. In *RelMiCS/AKA 11 2009*, Doha, Qatar, November 2009.

Type-Safe Distributed Programming for OCaml. John Billings, Peter Sewell, Mark Shinwell, Rok Strnisa. In *Proc. 2006 ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

Specifying and compiling Internet routing protocols

John N. Billings

Summary

The metarouting approach decomposes routing protocols into their linguistic and algorithmic components. A *routing language* specifies the form of metrics and policy, and the results of comparing metrics and applying policy to metrics, whilst a *routing algorithm* computes routing solutions to graphs labelled with a routing language. This dissertation describes a system that allows routing languages to be specified in a declarative style. These specifications can then be compiled into low-level code and combined with one of several pre-supplied routing algorithms to produce routing protocol implementations.

The system contains two types of routing algorithms: *offline* and *online*. Offline algorithms are variants of traditional shortest paths algorithms such as Dijkstra's algorithm, and allow the computation of routing solutions for static graphs. Online algorithms are generalised versions of current Internet routing protocols, obtained by removing the implicit routing languages and replacing them with *routing interfaces*. This latter type of algorithm allows the generation of fully-fledged Internet routing protocols.

Traditionally, routing protocols are monolithically specified and implemented. Specifications may reach hundreds of pages of informal prose, and implementations usually contain tens of thousands of lines of source code, often with serious errors. The metarouting approach has several significant advantages. Firstly, by separately specifying routing languages and reusing existing routing algorithms, it allows the rapid development of new routing protocols. Secondly, the system is able to determine whether a particular combination of routing language and algorithm is well-behaved. This involves the automatic inference of mathematical properties of routing languages. Finally, routing language specifications can be compiled into efficient code with minimal user intervention. In some cases the compiler is able to exploit inferred mathematical properties.

This dissertation concludes with an algebraic model of protocol interaction that is termed *simple route redistribution*. This model demonstrates how to avoid the safety problems that are associated with the traditional interaction mechanisms of *route redistribution* and *administrative distance*. Fundamental to simple route redistribution is a clear distinction between routing and forwarding. The model also captures the separation of *locators* and *identifiers* found in recent proposals for a new Internet architecture. It is hoped that this model will form a basis for incorporating practical methods for protocol interaction into the metarouting system.

Acknowledgements

I am very grateful to Tim Griffin, my supervisor, for his discussions and encouragement. This thesis would not have been possible without him.

I also thank the members of the metarouting research group: Md. Abdul Alim, Alex Gurney, Vilius Naudziunas, Balraj Singh and Philip Taylor. These people have made my time at the Computer Laboratory a much richer experience.

Finally, I thank Sarrita for her endless support and patience.

Contents

Chapter 1

Introduction

Consider sending a packet of data across the Internet. Prior to reaching its destination, the packet typically traverses a number of intermediate computers, known as *routers*. Each such router makes a local *forwarding* decision. That is, the router examines the packet, and based upon information contained within the packet, decides where it should next be sent. In most cases, the packet reaches its destination within perhaps a dozen or two of such hops [1]. The task of choosing and configuring these forwarding paths is termed *routing*, and it is with regard to this process that this dissertation is primarily concerned.

Routing on the Internet is complicated by the fact that it is a 'network of networks', with no central authority responsible for ensuring that paths are configured in a consistent manner. Moreover, the network topology is constantly changing: links are continually being added, removed, or even failing. Given this environment, how is it that the vast majority of packets successfully reach their intended recipients? The trick is to use a *distributed* approach to routing: each router constantly exchanges reachability information with its neighbouring routers, and in this manner builds up a map of the network. The exchange of reachability information is mediated by a *routing protocol*. Each router runs a program, called a *routing daemon*, that 'speaks' the routing protocol with neighbouring routing daemons. We further describe Internet routing, including the different kinds of routing protocols, in Section 1.1.

The Internet has been developed in an environment of 'rough consensus and running code' [2]. Whilst this approach has perhaps been pivotal to the success of the Internet, it has led to an assorted collection of monolithic routing protocols, each with its own quirks and peculiarities. This has several deleterious consequences.

- It is difficult to reappropriate components from routing protocol implementations. New routing protocols must often be built 'from scratch', with a large investment of labour. Therefore, network operators are instead coping with existing routing protocols and using them in situations for which they were never designed (§ 1.2). This causes unsafe behaviour, including routing oscillations and persistent loops.

- The mechanisms for allowing different routing protocols to interoperate are poorly documented and extremely primitive (§ 1.3). Again, this introduces the potential for unsafe behaviour
- Even within just a single protocol, it is possible to obtain seemingly-incorrect behaviour such as unintended paths or indeed unsafe behaviour. Many of these anomalies can be attributed to the unanticipated interaction of protocol features (§ 1.4).
- Routing daemons often contain implementation errors that cause them to crash, and can even lead to security vulnerabilities. This is perhaps not surprising given that protocol specifications are often several hundred pages of informal prose, with implementations containing tens of thousands of lines of source code (§ 1.5).

In light of the above, one might naturally conclude that the creation of a new routing protocol is a herculean task. How then can we help network operators and researchers to rapidly develop new routing protocols? One approach might be facilitate the adaptation of existing protocols. This is the approach taken by developers of the XORP routing platform. They have provided modular implementations of current routing protocols so that they can be more easily accommodated to new purposes. Whilst this is an interesting approach, the implementation of new routing protocols still remains a significant undertaking. Furthermore, there is no way of easily understanding the behaviour of protocols written using the XORP routing platform. We discuss the XORP architecture in more depth in Section 2.3.2.

In this thesis we adopt a radically different approach. We develop the seminal *metarouting* idea [3] to create a new architecture for the specification and compilation of routing protocols. Metarouting proposes that routing protocols be constructed by combining languages with algorithms. This separation of concerns appears pivotal to understanding the resulting routing protocols. The underlying theory of metarouting is based upon an *algebraic* approach to solving path problems [4, 5]. This theory has been advanced largely independently of the Internet engineering efforts, and instead is rooted in the more mathematical disciplines such as Operations Research. Using the metarouting approach, we create a high-level language for specifying routing protocols. Protocols defined in this way can then be *automatically* checked for correctness and *compiled* into efficient implementations. We introduce the algebraic approach in more detail in Section 2.6.

We now present our thesis that we argue for in the remainder of this dissertation:

> Using the theory of algebraic routing, routing protocols can be specified at a high level, automatically checked for correctness and then compiled into efficient implementations. The resulting protocols are easier to understand and require significantly less implementation effort.

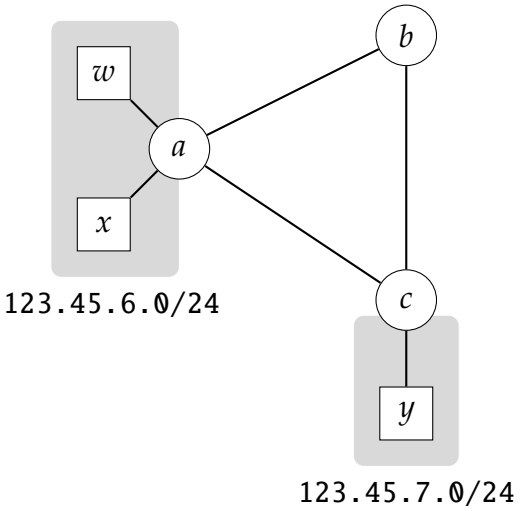


Figure 1.1: Example networks. Routers are drawn as circles and hosts as squares. Networks are depicted as shaded regions with associated prefixes.

The remainder of this chapter elaborates upon the motivations for this thesis. We commence with a brief introduction to Internet routing (§ 1.1). We then discuss how existing routing protocols are being used in situations for which they were never designed (§ 1.2), before describing the absence of mechanisms for safe protocol interaction (§ 1.3). Next we outline how protocol features can interact in unpredictable ways (§ 1.4) and examine the difficulties in implementing routing protocols (§ 1.5). Finally, we conclude with a summary of the contributions made by this dissertation (§ 1.6).

1.1 Internet routing

1.1.1 Routing and forwarding

In this section we describe the difference between routing and forwarding. This forms the basis for describing the different kinds of routing protocols. We illustrate the concepts using the example networks in Figure 1.1. Here there are two networks, with *prefixes* `123.45.6.0/24` and `123.45.7.0/24`. The first network contains two *hosts*, w and x, whilst the second network contains just a single host, y. There are also three routers, a, b and c. We note that a prefix in fact denotes a *set* of addresses [6]; we would expect the hosts in each network to be assigned addresses from within the associated network prefix. To simplify the presentation, we will assume that the names of hosts, such as w, are synonymous with their addresses.

Each routing daemon contains a set of *routes* stored in a data-structure known as a *Routing Information Base* (RIB). This is also informally known as a routing table. Routes result from exchanging reachability information with adjacent routers. We term this a

control plane process. In Figure 1.1, we would expect routers *a*, *b* and *c* to be running a routing protocol in order to exchange reachability information. A route abstractly comprises a triple (*prefix*, *metric*, *next-hop*), where *next-hop* is the address of the router to which traffic destined for *prefix* should be addressed, and *metric* is the cost of the associated path. Metrics are used by routing protocols to select between alternative routes to the same destination. The type of *metric* is dependent upon the particular routing protocol. Returning to Figure 1.1, router *b* might have a RIB containing the following entries:

Prefix	Metric	Next-hop
`123.45.6.0/24`	m_1	*a*
`123.45.7.0/24`	m_2	*c*

A router may be running several different routing protocols. The routes from each RIB are combined into a single data-structure known as a *Forwarding Information Base* (FIB) or forwarding table. The FIB controls the forwarding (or *data plane*) behaviour of the router. In the case of a router running just a single protocol, the FIB and the RIB contain similar information. Upon receipt of a packet of data to some address *d*, the router performs a process known as *longest prefix match* whereby it locates the most specific route in the FIB for that address. The data is then forwarded to the corresponding next-hop router (or directly to the destination, if it is located within the same network as the router).

Again returning to Figure 1.1, suppose that a packet of data arrives at *b* with a destination address of *x*. The longest-prefix match for that address is `123.45.7.0/24`, and hence the packet is forwarded to *a* (and then onto *x*).

1.1.2 Autonomous systems

Recall that the Internet comprises many individual networks. Networks can be grouped into *autonomous systems* (ASes), where each network within a given autonomous system is under the control of the same administrative entity. Figure 1.2 illustrates the concepts of autonomous systems and networks. Here there are two ASes, `AS1234` and `AS5678`. The first AS comprises the networks from Figure 1.1, whilst the second AS contains just the single network `234.56.8.0/24`. In practice, ASes often contain many more networks [7].

There are two different types of Internet routing protocols. An *interior gateway protocol* (IGP) is used for routing within an AS, whilst an *exterior gateway protocol* (EGP) is used for routing between ASes. Each AS is free to choose its own IGP, of which there are four in common use. The choice of IGP requires no coordination between ASes. We emphasise this point, because it means that we are able to develop new IGPs in the

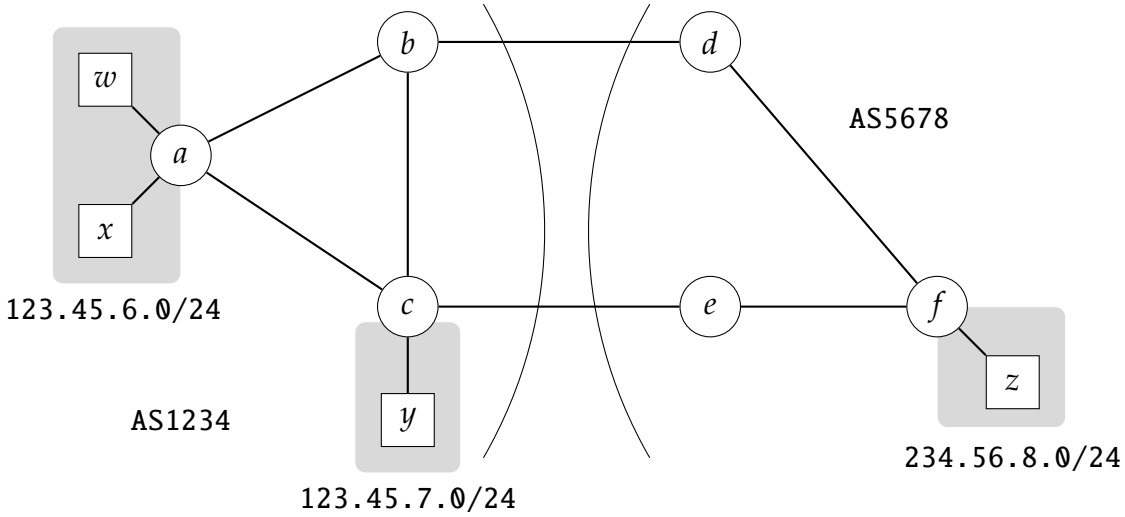


Figure 1.2: Example of two Autonomous Systems with constituent networks. Autonomous systems `AS1234` and `AS5678` are on the left and right, respectively.

context of a single AS and without requiring changes to the rest of the Internet routing infrastructure.

In contrast to the IGPs, there is only a single EGP, known as the Border Gateway Protocol (BGP) [8, 9]. The standardisation upon a single EGP for the whole of the Internet brings a large interoperability benefit for each AS. Returning to the example in Figure 1.2, the routing between the two ASes (i.e. between router pairs (b, d) and (c, e)) would be accomplished using BGP. Changing BGP would necessarily be a more long term goal, and therefore this is not the main focus of the dissertation.

1.1.3 Routing within autonomous systems

There are two main kinds of IGPs: *link-state/Dijkstra* and *distance vector*. A link-state routing protocol uses some underlying mechanism to distribute topology information between all routers within an AS. Each router can then run its own local routing algorithm, such as Dijkstra's algorithm [10] to compute paths to each destination prefix. The Open Shortest Paths First (OSPF) [11] and Intermediate System to Intermediate System (IS-IS) [12] routing protocols are examples of link-state protocols. Both of these protocols also contain mechanisms for further dividing an autonomous system into smaller routing domains, known as *areas*. This is necessary for larger networks due to scalability costs associated with topology distribution mechanisms.

A *vectoring* routing protocol instead sends *summaries* associating each network address with a cost. Both the Routing Information Protocol (RIP) [13] and the proprietary Enhanced Interior Gateway Routing Protocol (EIGRP) [14] are examples of distance vector routing protocols. Distance vector protocols can form temporary routing loops during

reconvergence (i.e. after a topology change) due to the presence of 'stale' information. EIGRP contains features from the Diffusing Update routing Algorithm (DUAL) [15] for coordinating routing changes in order to reduce such anomalies.

1.1.4 Routing between autonomous systems

Routing between ASes is accomplished using the Border Gateway Protocol. BGP is a mature protocol; the most recent recent version of BGP, BGP-4, was introduced in 1995. In common with RIP and EIGRP, BGP is a vectoring protocol.

There are two variants of BGP: external BGP (eBGP) and internal BGP (iBGP). eBGP is the 'standard' BGP, and is used for intra-AS, whereas iBGP is used for the distribution of eBGP-learnt routes within an AS. The use of iBGP means that external routes do not have to be redistributed into the IGP of an AS. Unless otherwise noted, we usually mean eBGP when we write BGP.

The BGP metric type comprises sets of *attributes* [8, 16]. The most significant attributes are as follows [16]:

Local preference This attribute is an integer value that denotes the per-router degree of preference for a route. Local preference values are only communicated internally to an AS, in iBGP-distributed routes.

AS path The AS path attribute comprises lists of AS numbers, denoting the ASes through which the route has been propagated. The AS path both prevents routing loops and is also used to compute a form of path length.

Next hop The next-hop attribute corresponds to the address of the router in the adjacent AS to which traffic for the associated prefix should be sent.

Multi exit discriminator This attribute, abbreviated to MED, is used to influence the degree of preference for a route *between* ASes. We further detail the MED attribute in Section 1.4.2.

Communities These are integer tags, whose meaning is configurable on a per-AS basis.

When a BGP router has several routes to the same prefix, it selects a single, 'best' route using the *route selection process*. This compares routes, attribute by attribute. Only if there is a 'tie' in the current attribute does the process move onto the next attribute. This form of comparison amounts to a lexicographic choice [17]. The order in which attributes are compared is as follows [16]:

1. User-configured policy. This stage may be influenced by communities.

2. Local preference; higher values are preferred.
3. AS path; shorter paths are preferred.
4. Multi-exit discriminator; lower values are preferred.
5. IGP distance to next-hop; lower values are preferred. This stage only applies to iBGP-learnt routes.
6. eBGP identifier; lower values are preferred. The BGP identifier corresponds to an IP address of the next-hop router. This stage only applies to eBGP-learnt routes.
7. iBGP identifier; lower values are preferred. This stage only applies to iBGP-learnt routes.

This section provides some insight into the rich policy that is possible using the BGP *policy language*. This is in marked contrast to the standard IGPs, for which metrics often only comprise a single integer value. The expressivity of BGP policy is necessary both for traffic engineering and also for implementing the complicated commercial contracts that govern the flow of traffic between ASes. However, as we show in Section 1.4, this degree of expressivity also introduces scope for unintended feature interaction. We now describe the different kinds of relationships that occur between ASes.

1.1.5 Economic relationships between autonomous systems

The economic relation between pairs of ASes can normally be categorised as being 'customer-provider' or 'peer-peer' [18]. In the former, the customer pays for the provider to carry the traffic of the customer. One example of this type of relation is found between commercial traffic carriers of unequal size. Here the smaller carrier pays the larger for transit facilities, and hence is the customer. A second, more common example occurs between an edge network, such as that found within a school or business, and an Internet Service Provider (ISP). Again, the edge network pays money to the ISP, and hence is the customer. In contrast, peer-peer relationships often indicate more equal roles between participating ASes. In the case of two carriers of equal size, this type of arrangement can be equally advantageous to both parties because a similar volume of traffic is exchanged in both directions. Therefore there is no exchange of money.

The type of commercial relation between a pair of adjacent ASes is often used to influence the BGP route selection process for routes learnt from each AS. Suppose that an AS has three routes to a prefix, occurring via a customer, a peer and a provider. Normally, the AS prefers to forward traffic using the customer route because such routes are a source of revenue. The next most preferred route is the peer route. These

routes are not a source of revenue, but also they do not require the AS to spend money itself. Finally, the least preferred route is the provider route. Here, the AS must pay the provider to forward traffic, and therefore it is desirable to minimise the volume of traffic sent over such routes.

1.2 The case for new routing protocols

1.2.1 BGP as an IGP

Many of the large ASes comprise geographically disparate networks, with some even spanning multiple continents. Current IGPs are ill-suited to such environments. Firstly, they exhibit poor scalability properties over large networks, especially where there are a large number of prefixes; it is for this latter reason that iBGP is used to distribute external routes into an AS. Secondly, current IGPs all compute variants of 'shortest paths', and have few facilities for administrative delegation. Instead, configuration must be centrally coordinated to ensure a consistent routing policy over an AS; altering a link weight in one part of an AS may have large effects on traffic flows in the rest of the AS. Such centralised control again does not scale, and may prove to be extremely challenging in larger ASes.

One solution has been to adopt eBGP or iBGP as an IGP, even though BGP was never designed for such purposes. Chapter 3 of [19] and Chapter 5 of [20] both provide advice on how to configure BGP for use as an IGP. One advantage of BGP is that it has much better scalability properties than current IGPs, although at the expense of greater convergence times. The protocol carries in excess of 300,000 prefixes on the wider Internet [1]. It also contains facilities for *aggregation*, whereby multiple prefixes are summarised under a single prefix. A second advantage of BGP is that it permits a large degree of hierarchical control over paths (although traffic engineering may still be problematic [21]). This allows individual ASes to be divided into multiple administrative regions, with local policy control delegated to each region. These ASes may come to resemble small internets in their own right.

There are also significant disadvantages with using BGP as an IGP. If the iBGP variant is used, then it is possible to obtain routing loops. As [19] ominously states, "the routing decision at every point in the iBGP cloud must be the same in order to prevent routing information loops". Such risks may appear to be an acceptable trade-off for the increased administrative control. However, there is also the opportunity to develop new BGP-like IGPs that do not have such drawbacks. This is one of the longer-term goals of the research presented in this dissertation.

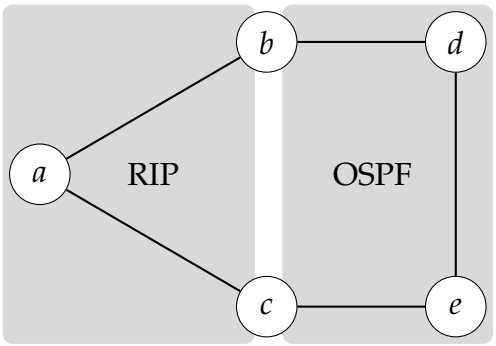


Figure 1.3: Example of RIP and OSPF routing instances. Shaded areas denote routing instances and circles denote routers. Neither routing instance contains reachability information from the other e.g. *a* cannot reach *d*. This problem can be addressed by using *route redistribution* at *b* and *c* to inject reachability information between instances.

1.2.2 Routing research

There is also broad scope for the development of new routing protocols in the context of network research; evaluation of proposed routing protocols cannot be entirely conducted under simulation. Instead, it is necessary to build implementations to empirically determine their properties. The difficulty is that building a new routing protocol requires a large investment of time and effort. Currently, the XORP platform may present one of the easier avenues for implementation. [22] references an example of a research group adding support for a wireless routing protocol to XORP. Turning to BGP, [23] describes a proposed replacement for BGP called the Hybrid Link-state and Path-vector protocol (HLP). This new protocol has 'vastly better scalability, isolation and convergence properties' than BGP. Again, the authors reference a prototype implementation built using XORP. The system presented in this thesis works towards further reducing the cost of implementation by allowing the compilation of routing protocols from high-level specifications.

1.3 The case for new glue logic

In this section we describe the current mechanisms for protocol interactions. We term this the 'glue logic', after [24]. Consider the situation in Figure 1.3. Here there are two *routing instances* [25, 26]. A routing instance comprises a connected collection of routing daemons running a given routing protocol; in the example there is a RIP instance and an OSPF instance. Routers *b* and *c* both run RIP and OSPF daemons whilst the remainder of the routers only run a single daemon. Routing instances do not inter-operate by default. Therefore router *a* cannot reach router *d*, for example.

The single mechanism for enabling interoperation between routing instances is *route*

redistribution. This is enabled on individual routers, and locally injects specified routes from the RIB of one routing daemon in the other. Returning to Figure 1.3, route redistribution could be enabled on routers *b* and *c* to inject routes from the RIP instance into the OSPF instance and vice versa – this is known as *mutual redistribution*. This would allow router *a* to reach *d*.

Note that it could be the case that two routing daemons on the same router contain routes to a common prefix. Indeed, mutual redistribution makes this situation more likely. For example, *b* could directly learn a route to *d* via the OSPF instance and also from a RIP route that has been redistributed from the OSPF instance by *c*. The FIB on each router selects between such routes using *administrative distance* (AD). This is a configurable value that is assigned to the routes from each routing daemon. The route with the lowest AD value is installed into the FIB.

Route redistribution is a valuable tool for connecting distinct IGP instances. For example, two companies might merge. Instead of reconfiguring the IGPs on each network, inter-connectivity could be established by using route redistribution. Route redistribution can also increase network reliability by eliminating dependence upon a single protocol. For example, should link (*b*, *d*) fail in Figure 1.3, then *b* could still reach *d* via the RIP instance. This is known as *domain backup* [27].

The problem with the route redistribution and administrative distance mechanisms is that they can cause forwarding loops and routing oscillations [24, 27, 28, 29]. Moreover, detecting where a particular configuration is susceptible to unsafe behaviour is NP-hard [28]. A set of *configuration guidelines* have been proposed to help prevent unsafe behaviour [27]. However, this does not address the fundamental problem: it should not be possible to configure routing protocols in an unsafe manner. Here we see the need for improved mechanisms for low-level protocol interoperation.

1.4 Designing routing protocols is difficult

When designing a new routing protocol, it is of the utmost importance that the behaviour of the specified protocol is well understood. Unfortunately, it is all too common to see widely-deployed routing protocols exhibit apparently contradictory behaviour due to the unanticipated interaction of features. In this section we give two examples of such behaviour. Both examples involve the BGP protocol.

1.4.1 Wedgies

Relatively recently, it was discovered that BGP can exhibit *unintended* non-deterministic behaviour [30]. This can lead to the existence of multiple stable forwarding solutions,

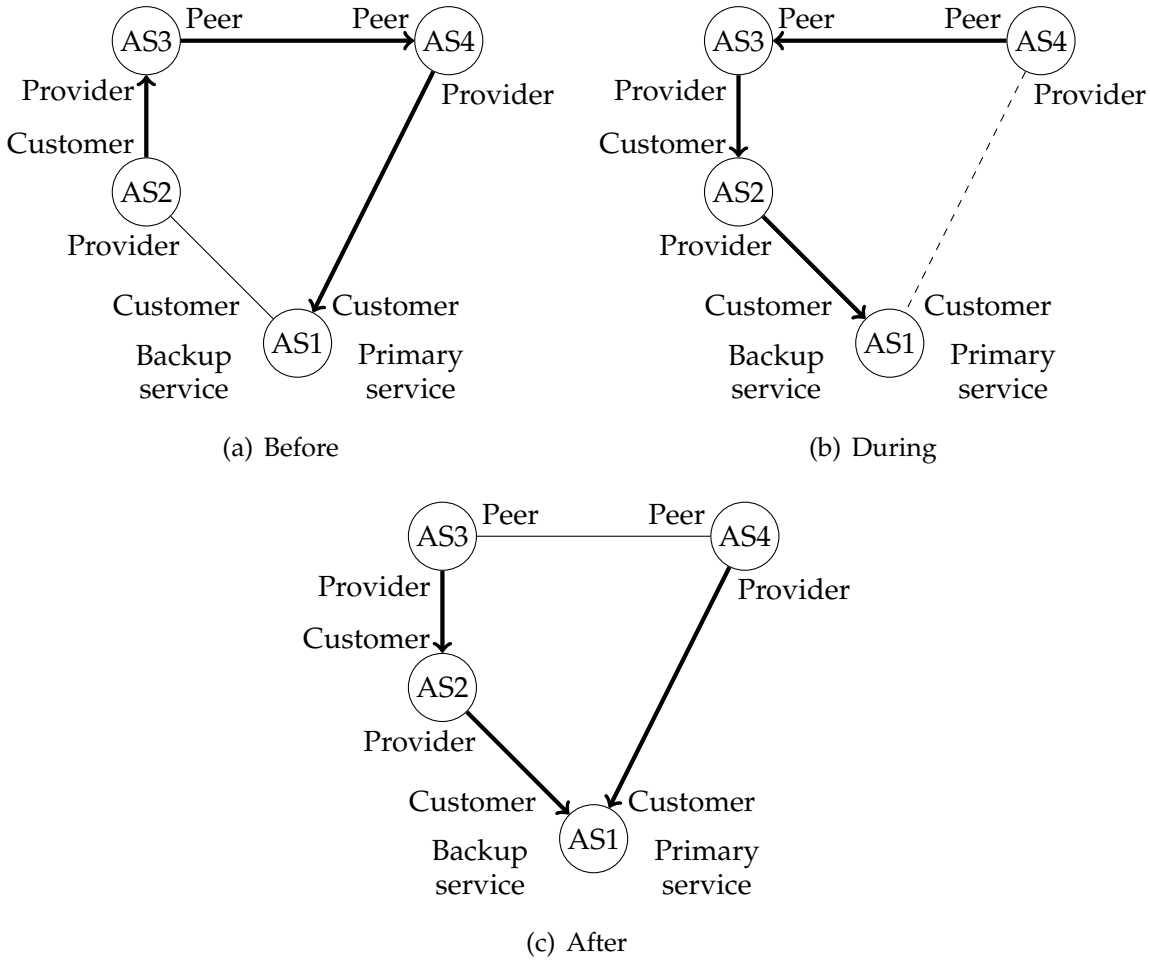


Figure 1.4: Example of a BGP wedgie

even though just a single such state is intended. Moreover, it is possible for the protocol to become 'wedged' in one of these unintended states, giving this problem its name. Coordinated operator intervention is required to diagnose the problem and restore the intended forwarding paths.

Figure 1.4, from [30], illustrates a BGP wedgie. In this example, there are four autonomous systems: AS1 – AS4. AS1 has two links to two providers: a *primary* link via AS4 and a *backup* link via AS2. The intention is that the backup link is only ever used if the primary link fails. This setup is commonly used in edge networks to increase reliability. There is also an additional autonomous system, AS3, that is both a provider to AS2 and a peer of AS4.

Providing that AS1 first announces its routes along its primary link then we have the intended situation, where no traffic flows over the backup link. This configuration is shown in Figure 1.4(a). If the primary link between AS1 and AS4 fails, then traffic switches to the backup link, again as intended. This second configuration is shown in Figure 1.4(b). The difficulty occurs when the primary link is restored. AS4 will announce the route 'AS4, AS1' to AS3. However, since AS3 prefers customer routes to

peer routes, as is the usual policy in BGP, then AS3 will continue to prefer the route 'AS3, AS2, AS1'. Furthermore, AS3 will not announce the primary route to AS2 because AS3 is itself not using that route. We therefore have the situation whereby AS3 and AS2 continue to route traffic over the backup link. This configuration is shown in Figure 1.4(c).

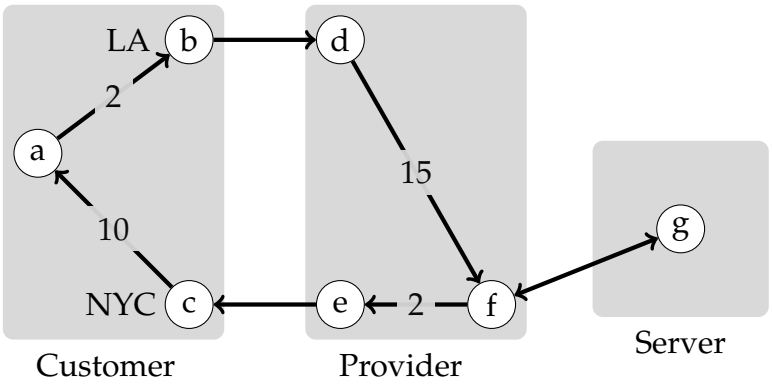
The network has therefore become 'wedged' in an unintended, but stable, configuration. Furthermore, AS1 cannot diagnose the problem without contacting AS3. From the view of AS1, the primary link has been restored, yet for seemingly inexplicable reasons a significant amount of incoming traffic remains on the backup link. The only way to restore the network to the correct configuration, shown in Figure 1.4(a), is for AS1 to temporarily bring down its session with AS2. This will cause all traffic to be routed over the primary link. Any traffic in transit may be lost whilst the network reconverges.

1.4.2 MED oscillations

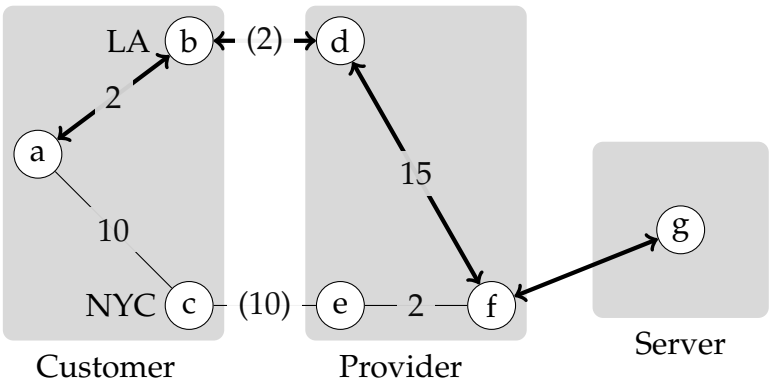
Another example of routing policy that can have undesirable effects is the BGP multi-exit discriminator (MED) facility. Consider a router within an AS that is forwarding traffic to a host outside of that AS. Usually, BGP will be configured so that the router will use the IGP to choose a path to the closest border router (or egress node) from the AS. This behaviour is known as 'hot-potato' routing because it causes ASes to hand off traffic as 'quickly' as possible. However, in some situations, it may be desirable for an AS to carry traffic for as long as possible on its own network before passing it onto a neighbour's network. This behaviour is known as 'cold-potato' routing.

One situation where cold-potato routing might be used is when a customer has constrained network resources. If the customer has multiple connections to their provider, then it might be possible for the customer to reduce resource usage on their own network by making traffic travel a greater distance on the provider's network. For example, consider the situation in Figure 1.5, which is taken from [31]. In Figure 1.5(a), the provider hands-off return traffic as quickly as possible, forcing the customer to carry the traffic a relatively long distance across their own network. In Figure 1.5(b), the customer uses the MED attribute to force the return traffic to choose an alternative egress point from the provider's network. The effect is that the traffic travels a greater distance on the provider's network and a lesser distance on the customer's own network.

A significant problem with MED is that it violates BGP's *independent ranking* 'rule'. That is, with MED enabled, the degree of preference for a route can depend upon the existence of other routes. A new route can prevent the current best route from being chosen, and not be chosen itself. For example, consider the situation illustrated in Figure1.6. In Figure1.6(a) there are two routes, r and s, to some common destination.



(a) Without MED, traffic always chooses closest egress point – 'hot-potato' routing. This can lead to asymmetric forwarding paths.



(b) With MED, the customer directs the provider to carry inbound traffic further on the provider's own network – 'cold-potato' routing. Symmetric forwarding paths are restored.

Figure 1.5: Motivating example for Multi-Exit Discriminator (MED) attribute. Customer uses MED to direct their provider to carry inbound traffic on the provider's own network, minimising usage of the customer's own network. Bold arcs are used for forwarding traffic. Inbound MED values are parenthesised, with lower values preferred.

Name	ASN	MED	IGP	Selected
r	AS1	_	10	
s	AS2	2	5	*

(a) Routes *r* and *s* are from different ASes and are therefore compared using IGP distances only.

Name	ASN	MED	IGP	Selected
r	AS1	_	10	*
s	AS2	2	5	
t	AS2	1	15	

(b) Routes *s* and *t* are from the same AS and are therefore compared first using MED values.

Figure 1.6: Example of route selection in the presence of the MED attribute. Routes from the same AS are compared first, with priority given to the MED attribute and then the IGP distance. The winning route(s) from each AS are then compared using IGP distances only.

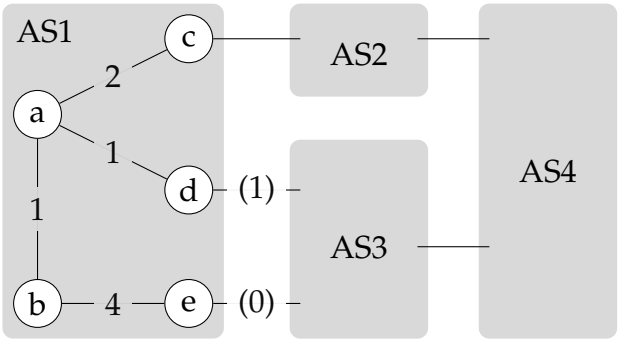


Figure 1.7: Example network configuration that is subject to a route oscillation caused by the MED attribute.

Suppose that r and s originate from different ASes. MED values are not compared between routes from different ASes and therefore route selection is purely based upon IGP distance. Hence s is selected due to its lower IGP distance.

Now suppose that AS2 originates a second route, t, illustrated in Figure1.6(b). Compared to the existing route from AS2, s, this new route has a lower (more preferred) MED value and a larger IGP distance. Since MED values are compared prior to IGP distances (but only between routes from the same AS), t is preferred to s. Now when t is compared to r, using the IGP attribute, r is selected. The introduction of t has caused the relative preference between r and s to become inverted.

This violation of BGP's independent ranking rule complicates the route selection process, potentially leading to operator error. Without MED, it is sufficient to compare each route with the current best route, keeping the new route if it is better than the current one. This amounts to ordering the routes, and then choosing the best. However, with MED the ordering of routes is affected by the presence of other routes, and hence this route selection algorithm is no longer sufficient.

There is also a more serious problem with MED: it can also introduce *route oscillations*, whereby the preferred route continually changes [32]. Oscillations are undesirable because they can cause datagrams to be lost. Furthermore, the continual routing updates consume network resources. Note that unsafe behaviour in BGP is also possible without using the MED attribute [33, 34].

Figure 1.7 illustrates a route oscillation. This example and the following argument are taken from [31, 35]. Routers a and b are BGP *route reflectors*. That is, they redistribute routes between the client routers c, d and e. Let r_c, r_d and r_e be routes to AS4 via c, d and e respectively. Consider which route is chosen by a to reach AS4. Assuming that a always learns of r_c and r_d, then there are two cases to consider. In the first case, a additionally learns of r_e and therefore chooses r_c due to a combination of MED values and IGP distances. However, for a to learn of r_e requires b to choose r_e. This is a contradiction, because b would learn of r_c and r_e, and hence would select r_c due to IGP

distances. In the second case, a does not learn of r_e and therefore chooses r_d due to IGP distances. However, for *a not* to learn of r_e requires b to select r_c over r_e. Again, this is a contradiction because b would learn of r_d from a and hence would select r_e due to its lower MED value.

We see that there is no stable solution. The practical implication is that the network configuration would continually oscillate, with a and b perpetually exchanging routing updates. This example illustrates how an apparently minor addition to a routing protocol can have unforeseen and indeed undesirable consequences.

1.5 Implementing routing protocols is difficult

Suppose that we have a routing protocol design that satisfies a high-level specification. There still remains a significant challenge in correctly implementing the specification. Errors made during implementation can lead to protocols that exhibit unexpected behaviour, such as selected routes not actually being used to forward traffic, or daemons that crash when presented with particular inputs. The end-result is a decrease in network reliability.

Interesting examples of protocol implementation errors can be found in the Quagga [36] BGP release notes. Note that Quagga contains probably the oldest open-source implementation of BGP, having supported the protocol for the past decade, and therefore we might hope that most bugs would have been discovered. Furthermore, there are numerous reports of the Quagga BGP daemon being considered sufficiently stable for use within smaller networks. The 0.99.10 version of Quagga, released on 11/06/2008, includes the following bug-fixes:

```
[bgpd] Bug #419: partial aspath-limit incorrectly causes session reset
[bgpd] Fix crash on startup if compiled IPv4-only
[bgpd] Fix number of DoS security issues, restricted to configured peers.
[bgpd] Small fix for crash if 'listenon' argument is not given
[bgpd] Fix typo, which prevented advertisement of MP (non-IPv4) prefixes
```

The first bug appears to cause a session reset between routers if the AS path attribute exceeds a particular length. The next three bugs can each cause the BGP routing daemon to crash. The last bug can cause multi-protocol prefixes not to be advertised. Each of these bugs will lead to a loss of connectively for any hosts using routes that have been advertised by the affected routers. Examining the release notes from other Quagga versions, it is clear that such bugs are relatively common.

There are also several documented occasions in which routing protocol implementation errors have caused large-scale loss of Internet connectivity. In February 2009, an erroneous BGP announcement containing an AS path with 252 entries triggered a bug

in the Cisco BGP implementation [37]. As the BGP route propagated across the Internet and the AS path length reached 255, it caused the affected BGP daemons to crash. During this event, the connectivity to an estimated 4.8% (12,920) of all prefixes either changed or failed [38]. This is an approximate ten-fold increase in the standard level of prefix instability. In May 2009, a routing update with a particular 4-byte AS number triggered a bug in older versions of the Quagga BGP implementation, including version 0.99.10 [39]. Again, the routing update caused the affected BGP daemons to crash, leading to a loss of connectivity for over 1000 prefixes.

One high-level source of errors can be attributed to the degree of complexity in protocol specifications and implementations. Again returning to BGP, we see that the plain BGP-4 specification [9] runs to one hundred and four pages, much of which is informal prose with potential scope for interpretation error. This relatively large specification translates to relatively large code-bases. For example, we list the number of lines of code in recent versions of three open-source BGP implementations:

Daemon	Version	Lines of code	Language
OpenBGPD [40]	4.4.1	20,140	C
Quagga BGP [36]	0.99.12	55,276	C
XORP BGP [22]	1.6	30,721	C++

We see that none is particularly small, with Quagga exceeding fifty-five thousand lines of code. Inevitably, larger code-bases are more difficult to reason about, and hence have greater scope for implementation error.

A confounding source of implementation errors involves the *kind* of code required to implement a routing protocol. A routing daemon contains interfaces to at least three different domains, each with its own associated language and semantics. Firstly, there is the administrative interface which is used to configure the router. Secondly there is a network interface which is used to communicate routing updates between adjacent routing daemons. Finally there is the forwarding information base interface which is used to add and remove routes from the host's forwarding table. Each of these interfaces can modify a large amount of internal state, such as routing tables, network interfaces and policy. Implementation errors can introduce inconsistencies between these different kinds of state. There is also the associated difficulty that each of these interfaces can be accessed at the same time, introducing scope for concurrency errors.

We therefore see that the task of implementing a new routing protocol is a significant undertaking. Furthermore, the resulting implementation is likely to contain errors which compromise the reliability of the router. Seen in this light, it is not surprising that new routing protocols are rarely developed.

1.6 Contributions

The main contribution of this dissertation is to demonstrate the feasibility of defining routing protocols in terms of their linguistic and algorithmic components. This contrasts with the current practice of *monolithically* specifying and implementing protocols. We show that there are multiple benefits from our new approach. Firstly, the routing language component is amenable to automatic verification, eliminating a large class of errors that may be found in current protocol designs. Secondly, we show the feasibility of abstracting current protocol implementations to obtain generic, well-tested algorithms. Finally, we demonstrate that the algebraic properties that are automatically determined during routing language verification can be used to produce more efficient protocol implementations.

The secondary contribution of this thesis is an algebraic model of protocol interoperation. The disadvantage of current algebraic models is that they are only applicable to *single* protocols, whereas our approach demonstrates how to model the behaviour of *multiple* protocols. Whilst our model is rather abstract, it is a promising approach to this difficult problem. Our model also demonstrates an interesting new use of algebraic structures called semi-modules.

The specific contributions of this thesis are as follows:

Routing algebra metalanguage (RAML) We have defined an algebraic, domain-specific language for the specification of routing languages (that is, a *meta*-language). This language is capable of specifying many of the idioms found in current routing protocols, such as lexicographic choice and zoned routing. This language is based upon an earlier, less expressive version found in [3], and contains features from [17].

Library of routing algorithms We have generalised a number of open-source routing protocol implementations from the Quagga routing suite [36] to obtain *generic* routing algorithms. These can be combined with compiled RAML specifications to produce new protocol implementations. We have also defined a generic *routing interface* between routing algorithms and compiled code. This work has been done in collaboration with Philip Taylor and Md. Abdul Alim.

Compiler architecture We have designed and implemented a compiler architecture from compiling RAML specifications to executable code. This includes a framework for the automatic derivation of algebraic properties and an *intermediate routing language* (IRL) for compactly specifying the semantics of RAML specifications. We demonstrate how to embed IRL expressions into the C++ template language, eliminating the need for low-level code generation.

Optimisation of generated code The automatic verification infers algebraic properties of the IRL expressions. We show how to reuse these properties to generate more efficient code.

Semimodule model of protocol interaction Our semimodule model of protocol interaction abstractly models current idioms such as hot-potato and cold-potato routing. It also captures the separation of *locators* and *identifiers*, as found in recent proposals for a new Internet architecture [41]. Fundamental to this model is a clear separation between routing and forwarding.

We commence with an overview of background and related work in Chapter 2. We then discuss the system architecture in Chapter 3. Next, in Chapter 4, we describe the semantic domain upon which we base our metalanguages. We then describe a restricted metalanguage, RAML_1, in Chapter 5 and describe how to compile it in Chapter 6. In Chapter 7 we describe RAML_2, which is a more expressive version of RAML_1. In Chapter 8 we show how to measure and, using properties automatically inferred during compilation, increase the performance of compiled code. We then present an algebraic model of protocol interactions and route redistribution in Chapters 9 and 10. Finally, we summarise the results of this thesis and outline areas for future work in Chapter 11.

Chapter 2

Background and related work

In this section we introduce the background and related work which this dissertation builds upon. We commence with an overview of the history of Internet routing (§ 2.1). We then review applications of *domain-specific languages* to networking, before summarising previous work towards modularising the control plane (§ 2.3), the data plane (§ 2.4) and network protocols (§ 2.5). Next we discuss routing algebras (§ 2.6). These form the mathematical basis for our specification language which we use to describe the linguistic component of routing protocols. Finally, we describe several routing algorithms (§ 2.7). The metarouting system combines compiled routing languages with routing algorithms to produce routing protocols.

2.1 A brief history of the Internet

One of the earliest visions of the Internet can be attributed to J.C.R Licklider, of the United States Department of Defence Advanced Research Projects Agency (DARPA) [42]. In 1963 Licklider published his 'Memorandum For Members and Affiliates of the Intergalactic Computer Network' [43], which outlined ideas for network-based data access and remote execution of programs.

Two years previously, in 1961, Leonard Kleinrock submitted his PhD thesis proposal 'Information Flow in Large Communication Nets' [44] which outlined a plan of study of the theoretical properties of packet switched networks (PSNs), which form the basis of the current Internet. This type of network characteristically segments data into uniform messages, termed 'packets', which are then copied from node to node within the network until they reach the destination [42]. Significantly, packets are independently routed at each node. Kleinrock subsequently published several papers that made pioneering contributions to the theoretical understanding of PSNs, demonstrating, for example, that packet switched networks could be more efficient than circuit switched networks (which reserve dedicated paths, or 'circuits', for each connection).

The contributions of Licklider and Kleinrock led to the creation of the first packet switched network in 1969, called the ARPANET [42]. This network grew into today's Internet as other networks were incrementally connected to it. Central to the ARPANET were packet switches, termed Interface Message Processors (IMPs) [45]. These are precursors of current Internet routers. In contrast to today's Internet, the network itself attempted to reliably deliver traffic. However, the ARPANET did include a dynamic routing protocol based upon a distance vectoring algorithm. Paths were selected so as to minimise the total transit time of packets This required that link weights were continuously adjusted based upon output queue lengths. The status of the ARPANET was changed from an experimental network to an operational network in July 1975. At this time there were fifty routers and more than seventy hosts.

The prototypical Internet protocol suite, which later became today's Transmission Control Protocol/Internet Protocol (TCP/IP) was first defined in 1974 [46, 47]. This model made several fundamental contributions. Firstly, it introduced the notion of a gateway as an interface between networks, and described its role as a mechanism for translating between networking technologies. Secondly, it described the need for a uniform addressing scheme that was independent of any particular network addressing scheme. Finally, it outlined an *internetwork protocol* for reliable stream-based transmission *between* networks in the presence of different maximum transmission sizes and time delays. Prior to this point, researchers had only addressed the problem of communication within single networks. The described protocol contained many of the features found in today's TCP/IP, including segmented transmission, the use of sequence numbers with a 'sliding-window' to detect data loss and duplication, and a time-out based retransmission strategy. This monolithic protocol was soon separated into individual IP [48] and TCP components [49].

In the late 1970s the ARPANET adopted a new internal routing algorithm called the Shortest Path First (SPF) algorithm [50]. The primary motivation was the poor scalability with the distance vector routing scheme, especially as the ARPANET now comprised about a hundred routers. The scaling issue may have been caused by the high frequency of routing updates; the interval between updates was just 128ms. A secondary motivation for changing the routing protocol was the observation that the distance vectoring algorithm caused temporary, network-wide cycles during reconvergence (recall that BGP prevents this problem by including AS paths within routing updates).

The SPF algorithm comprised two components. Firstly, there was a link-state flooding mechanism, where each router periodically sent the local status of its links to all other routers. These updates were typically smaller than the previous distance vector updates, and required less processing. Note that SPF still dynamically computed link weights based upon delays. Secondly, each router used the link-state updates to construct a weighted graph of the network. Each router then used Dijkstra's algorithm [10] to independently compute shortest paths across the graph. Today's OSPF [11] and IS-

IS [12] IGPs use a similar scheme.

Starting at the end of the 1970s, TCP/IP was used to start connecting the ARPANET with other nascent networks. The host protocol of the ARPANET was itself switched from the original Network Control Protocol to TCP/IP on January 1st 1983 [51]. Inter-network routing was performed using the Gateway to Gateway protocol [52], which was based upon the initial ARPANET distance vectoring protocol. A significant problem with GGP was that it had no notion of autonomous systems. Instead, the whole of the Internet formed a single routing domain, with all gateway nodes participating as equals within the routing algorithm (GGP later became viewed as an IGP [53]). This led to poor scaling and maintainability. Further, since every gateway was 'trusted', a single malfunctioning gateway could cause Internet-wide traffic disruptions [54].

The problems with GGP were partially addressed by the development of the Exterior Gateway Protocol (EGP) in 1982, a 'standard for Gateway to Gateway procedures that allow[ed] the Gateways to be mutually suspicious' [54]. EGP was the first interdomain routing protocol, incorporating the concept of independent autonomous systems. Just as today, an autonomous system was able to run its own internal routing protocol which would be largely unaffected by external failures. EGP was used to exchange reachability information between gateways within both the same AS and also with those in neighbouring ASes. One limitation with EGP was that it assumed a tree topology of ASes built around a single 'core' network [55]. This requirement was necessary because there was no facility for detecting loops.

Throughout the 1980s, the Internet continued to expand. For example, the Computer Science Network (CSNET) [56] was created in the early 1980s. CSNET was intended to be accessible for all computer science researchers in the United States. The network was connected to the ARPANET and used the TCP/IP protocol. One problem with such networks was that they were aimed at specific research communities. This motivated the creation of the National Science Foundation network (NSFNET) in the mid-1980s as a network for *all* academics [42]. The NSFNET proved to be extremely successful, and replaced the ARPANET backbone in the late 1980s.

The tree topology imposed by EGP was soon recognised as a severe restriction to the growth of the Internet [57, 58]. The Border Gateway Protocol [59] was developed in the late 1980s to address this issue. BGP removed the need for a distinct backbone, and instead permitted arbitrary topologies of peer ASes. This led to a less hierarchical Internet topology. The original version of BGP mandated up/down/horizontal relations between ASes. This requirement was removed in BGP-2 [60]. The most recent version of the routing protocol, BGP-4 [8], was defined in 1995.

Throughout the 1990s, the Internet continued to grow as increasing numbers of networks were connected. During this period, an increasing proportion of Internet traffic was of a commercial nature, often using private infrastructure. This led to NSFNET

playing a less pivotal role for the rapidly developing Internet. The NSFNET backbone was finally retired in 1995 [61]. At this point there were millions of users, and the Internet had largely assumed the architecture that remains to this day.

2.2 Domain-specific languages for networking

Our system accepts protocol specifications in a purpose-built language called RAML (Routing Abstract MetaLanguage). Additionally, 'under the hood' in the protocol compiler there is an algebraic language embedded within the C++ template language. Both of these languages are examples of *domain-specific languages* (DSLs). In this section we informally define what constitutes a DSL and show how particular networking problems, such as packet parsing, can benefit from DSLs. We also briefly discuss implementation techniques.

2.2.1 Overview

A domain-specific language is a programming language that is tailored to solving problems in a particular problem area [62]. For this reason, DSLs typically have fairly restricted syntax when compared to 'general purpose' languages. Furthermore, DSLs are often *declarative*. That is, they specify what problem is to be solved without defining the particular manner in which it is to be solved, leaving the DSL compiler free to select the most appropriate method. Hundreds of DSLs have been created for areas ranging from the specification of financial contracts [63] to graphics, animation and simulation [62]. Mainstream examples of declarative DSLs include Yacc [64] for writing parsers, Make [65] for specifying build rules and LaTeX [66] for writing documents.

The main benefit of DSLs over general purpose languages is that the semantics of DSLs can be better matched to the particular areas of application. Therefore programs written in DSLs can be more concise and hence easier to understand than their counterparts written in general purpose languages. Furthermore, DSL compilers can utilise domain knowledge for purposes such as increasing efficiency and providing automated error handling. As we show in Section 2.2.2, Yacc provides an example of a DSL that is highly-tailored to the problem of parsing. Moreover, parser specifications written in Yacc can be compiled into efficient, low-level code with automated error-handling.

Compilers or interpreters for DSLs can be implemented in the same manner as those for general purpose languages. Another, alternative method is to *embed* the DSL within a *host* language – resulting in a domain specific embedded language (DSEL) [67]. In practice, this means defining datatypes within the host language to represent the DSEL. A parser, evaluator and pretty-printer can then be written within the host language to

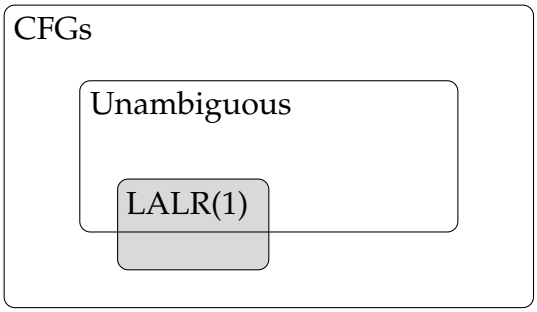


Figure 2.1: Relationships between context-free grammars (CFGs), unambiguous grammars and LALR(1) grammars of Yacc

process expressions within the domain-specific language. The benefit of this approach as opposed to writing a stand-alone interpreter or compiler, is that programs can be written using a combination of the host language and the DSEL.

2.2.2 Yacc as an analogy for metarouting

In this section we describe the Yacc DSL for parser generation, and compare it with our system for generating routing protocols. Parsers written in Yacc are typically much shorter than those that have been hand-written. Yacc is also able to generate automatic error-handling code. Similarly, the metarouting system can generate entire routing protocols from specifications that are typically a few lines long. Furthermore, the generated protocols contain code for automatically handling errors such as invalid configuration.

Yacc is based upon the theory of context-free grammars (CFGs). However, this formalism is only implicitly represented in the Yacc language. Similarly, the metarouting system is based upon the theory of algebraic routing. Internally, metalanguage specifications are translated into mathematical objects such as binary operators and preorders, although these formalisms are not explicitly represented in the metalanguage.

Yacc may only be used to write parsers for the LALR(1) subset of context-free grammars (CFGs), as shown in Figure 2.1. LALR(1) grammars are so named because their associated parsers can always be defined so as to operate from *Left to right* on the token input stream (i.e. initial token first), producing a *Right-most* derivation. Furthermore, such parsers can always select the next production rule using at most *one* token of *Look-Ahead*.

Therefore, whilst Yacc cannot be used to express all CFGs, it can still express a very useful subset. For example, the syntax of many programming languages can be expressed as LALR(1) grammars (C++ is a notable exception). Analogously, not all routing algebras may be expressed within our metalanguage. However, we believe that those that it can express are still of practical use.

Some CFGs have the property that they are *ambiguous*. That is, a string in the language of the grammar may be generated in more than one way. When parsing with an ambiguous grammar, it may be possible to reach a situation whereby there is a choice of several possible parse rules, and hence an error must be raised. We note that *Generalised LR* (GLR) parsers evade this restriction by performing *all* possible parses, providing that they amount to a finite number; whenever there is a choice of reduction rules to apply, evaluation proceeds by separately applying each such rule 'in parallel', thereby performing a breadth-first search of the space of possible parses.

Some LALR(1) grammars are ambiguous, as illustrated in Figure 2.1. Hence it is possible to write an ambiguous parser in Yacc, although it is possible to detect the ambiguity when compiling the parser specification. Analogously, within our system it is possible to specify routing languages that do not have the required algebraic properties. In common with Yacc, we raise an error at compile-time when such situations are encountered.

LALR(1) grammars admit efficient, table-based parsers. Yacc translates parser specifications into this form. The Bison variant of Yacc also supports the aforementioned GLR parsers, although it only produces optimised implementations for the LALR(1) case; whilst the LALR(1) parsing algorithm has linear worst-case complexity, the current implementation of the Bison GLR parsing algorithm can require exponential time and space in certain situations. Within the metarouting system, we are able to generate more efficient implementations for a restricted set of routing algebras that have particular algebraic properties.

2.2.3 Packet parsing

There has been considerable research devoted to the problem of language techniques for generating efficient, error-free packet parsing code. PacketTypes [68] is a DSL for specifying packet formats. This system eliminates the error-prone process of manually implementing packet parsers (often in C). Furthermore, PacketTypes specifications retain much of the semantic information contained in textual packet format specifications. PacketTypes specifications are compiled into efficient C code which automatically checks for errors. The resulting code can be interfaced with other C code.

BinPAC [69] is another DSL that is similar in scope to PacketTypes. However, BinPAC is targeted towards analysis of application-level traffic for the Bro intrusion detection system. BinPAC specifications are able to represent bi-directional connection state. This is necessary to handle protocols such as HTTP where replies must be associated with particular requests. Furthermore, the language supports incremental parsers for concurrent input. BinPAC specifications are compiled into C++ code.

Several other DSLs have been created for specifying parsers for more general data

formats. For example, the DataScript [70] language has been used to specify parsers for both the ELF object format and also Java class files. More recently, PADS [71] has been created for specifying parsers for ad-hoc data. PADS specifications are based upon dependent types. The systems includes facilities for visualising and querying data sources.

The metarouting system itself generates parsing code for route metrics. Whilst it might be possible to use the aforementioned DSLs, we found that it was straightforward to automatically generate parsers ourselves from protocol specifications.

2.2.4 Packet processing

Within the context of DSLs, packet *processing* has received far less attention than packet parsing. Packet processing typically involves high-speed network switching, and is therefore a more specialised application than plain packet parsing. With the advent of high-performance network processors (NPs), such as the Intel IXP2400 [72], implementing packet processing applications has become increasingly complicated. Modern NPs comprise multiple processor cores with a distributed memory architecture. NP applications normally move packet data between multiple *memory banks* as it is processed through a pipeline of cores. Safely managing memory in this environment is tedious and error-prone.

One possible solution to programming NPs is the PacLang DSL [73]. This is an imperative language for multi-threaded systems that can be targeted to NPs. The language includes a linear type system for ensuring memory safety. A case-study demonstrates that the type system may be particularly suited to packet processing applications. Whilst PacLang remains a research-level language, it demonstrates that DSLs can be applied to areas that are traditionally viewed as low-level systems programming.

2.2.5 Declarative networking

P2 is a system for expressing distributed overlays as high-level, declarative specifications [74]. The system is based upon the NDlog language, a network-oriented version of the Datalog logic programming language. The authors report that some overlay implementations require two orders of magnitude less code when implemented within P2. It is also possible to express routing algorithms within P2. NDlog specifications are compiled into programs for a distributed runtime system that uses a dataflow architecture.

We view P2 as a possible target architecture for the metarouting system. There is no guarantee of correctness for routing protocols written within P2, and therefore it would still be necessary to retain the property-inference aspect of the compiler.

2.3 Extending the control plane

In this section we discuss approaches to modularising the control plane. We first discuss the open-source Quagga and XORP routing platforms, before briefly reviewing the closed-source routing architectures.

2.3.1 Quagga

Quagga [36] is one of the more mature, open routing platforms. It contains implementations of the OSPF, BGP, RIP and IS-IS routing protocols. Each routing daemon runs as a separate process, increasing the reliability of the system. Quagga also contains a central daemon (zebra) that coordinates the individual routing daemons and communicates with the operating system routing interface. Quagga is built around a central library of code that is shared by the different protocol implementations.

The metarouting system contains a number of algorithms that are generalised versions of the Quagga routing protocols. We discuss the process of generalising routing protocols in Chapter 3. Whilst we have not added any *new* routing protocols to Quagga, we have found it relatively easy to *modify* the existing protocols.

2.3.2 XORP

The XORP router platform was born out of research to design low-latency, extensible router software without sacrificing efficiency [22]. The developers argue that current problems with high convergence times and limited innovation are not inherent to the Internet architecture or the design of routing protocols. Rather, such problems can at least be partially attributed to the architecture of current routing protocol *implementations*. Many such implementations focus upon scalability at the expense of latency and extensibility.

The goals of extensibility and latency have led the XORP designers to adopt several non-standard architectural features. Firstly, each protocol within XORP is implemented as a separate process. This design provides a high degree of isolation between different routing daemons; an error in one daemon is less likely to affect another. The multi-process approach contrasts with the more usual design of having a single process for all protocols. Quagga is notable as the only other open-source router platform to also have a multi-process design. Interestingly, Cisco switched from a uni-process architecture in their IOS platform to a multi-process architecture in their newer IOS XR platform.

Routing platforms typically supply facilities for coordinating different protocols. For example, BGP hot-potato routing is dependent upon IGP distances, and therefore requires that there is a mechanism for the RIB to supply selected IGP routes. A second

example is route redistribution. Here the RIB must pass routes between (almost) arbitrary pairs of protocols, and without impacting local route selection. In both of these situations, the coordination mechanisms must be efficient. The co-ordination of protocols is easier to achieve in a uni-process architecture where, due to the single address space, it is easy to share data-structures between different protocols. In common with Quagga, the multi-process design of XORP has required the creation of an efficient *inter-process communication* (IPC) coordination mechanism.

Another non-standard design adopted by XORP is an event-driven routing pipeline. A more common technique, used by both Cisco and Quagga, is to use a scanner mechanism to periodically process pending route updates. The disadvantage of this approach is that it causes high latencies. In contrast, an event-driven architecture processes updates as soon as they are ready. This facility is more complicated to implement in a multi-process architecture due to the coordination requirements. For example, in the case of BGP, an IGP event may require the notification of the BGP daemon, which will in turn cause a BGP event.

XORP achieves modularity within routing protocols by dividing routing into a number of distinct stages, with each stage containing a table of routes. A routing daemon then comprises a number of tables that have been appropriately wired together. There is some overhead in terms of time and memory when compared to the conventional 'one big table' approach. However, the advantage is that protocol implementations become more modular and hence easier to modify. Moreover, events such as a BGP peering going down can be easily handled by dynamically inserting new stages into the BGP pipeline.

It is difficult to evaluate the degree of extensibility that XORP provides when compared to Quagga, for example. Whilst it is possible that the XORP architecture may indeed admit a more modular and hence less error-prone implementation, there is little evidence to evaluate this claim. However, in the longer-term we are certainly looking to generalise the XORP routing protocols for inclusion into the metarouting system.

2.3.3 Proprietary routing platforms

The main providers of proprietary routing platforms provide few mechanisms for modifying or extending the routing software. Juniper has released the JUNOS SDK which provides interfaces for packet manipulation on Juniper routers. Illustrative examples of services created using the JUNOS SDK include load balancing, traffic classification and quality-of-service monitoring [75]. The ExtremeXOS router operating system from Extreme Networks also has limited facilities for extensibility, such as XML interfaces and TCL scripting [76]. In common with the JUNOS SDK, these features appear to be mainly targeted at placing additional services on routers and switches,

instead of permitting modifications to routing daemons.

2.4 Extending the data plane

In this section we describe approaches towards building extensible data planes. We view this area as complementary to the metarouting system.

2.4.1 Click

Click [77] is a system for building modular, high-performance forwarding planes. Typical applications are network devices such as routers and traffic shapers. A Click forwarding plane comprises a number of *elements*. An element performs a single, well-defined action, such as decrementing the Time to Live (TTL) value of an IP packet header. A pair of elements can be associated using a *connection*, which represents the potential for the directed transfer of packets from one element to another. A forwarding plane is constructed by connecting elements to form a directed graph.

Relatively sophisticated traffic scheduling policies can be performed by combining elements. For example, the developers of Click have implemented several variations of Random Early Drop, which is a scheme for reducing network congestion by dropping packets when link buffers start to become full. They have also implemented all of the functions from the Differentiated Services (Diffserv) architecture. Components include packet classifiers, which tag packets as they enter networks, and traffic shapers, which control packet rates within networks. These examples demonstrate that the Click architecture is both flexible and expressive.

Click is relatively performant compared to the standard Linux forwarding plane; one simple benchmark shows that a Click-based router has a maximum loss-free forwarding rate (MLFFR) of three times that of a pure Linux router. One reason for this difference is that the Click device handling elements use direct polling of device DMA queues instead of interrupts. This prevents *receive livelock* where increasing numbers of packets cause the system to spend the majority of its time servicing interrupts at the expense of forwarding packets.

2.4.2 NetFPGA

The NetFPGA system [78] comprises a PCI card with a Field-Programmable Gate Array (FPGA) and four gigabit ethernet interfaces, as well as supporting software. It is targeted at building high-performance, reconfigurable data-plane components. An

FPGA is a programmable logic device; the use of an FPGA allows line-rate forwarding that would not be possible using a standard ethernet interface on a desktop computer.

We view NetFPGA as an ideal counterpart for the routing protocols generated using the metarouting system; the NetFPGA authors have already demonstrated how to combine the NetFPGA with routing platforms such as Quagga or XORP to build routers with high-performance data planes. Currently our generated protocols are based upon Quagga routing daemons, and therefore we believe that it would be straightforward to also combine them with NetFPGA.

2.4.3 OpenFlow

OpenFlow [79] is a project that aims to help build reconfigurable ethernet networks. OpenFlow comprises a standardised, open protocol for viewing and modifying the flow state in ethernet switches. This permits individual flows to be sampled or rerouted, for example. Furthermore, experimental and production traffic may co-exist on the same physical infrastructure. The authors currently provide a reference OpenFlow ethernet switch built using the NetFPGA system, and are also aiming for ethernet switch manufacturers to incorporate the OpenFlow interface into their products.

OpenFlow is again complementary to the metarouting system. OpenFlow ethernet switches may facilitate experimentation with routing protocols generated using the metarouting system. For example, experimental traffic could be routed using generated routing daemons, whilst production traffic could remain unaffected.

2.5 Modularising network protocols

In this section we discuss approaches towards building modular network protocols. Again, this area is complementary to the metarouting system, although we share some of the techniques.

2.5.1 Ensemble

Ensemble [80] is a 'group communication' system for building distributed applications. Ensemble is implemented in ML, and comprises a number of protocols that can be composed to build networking stacks. [81] shows how to formalise components from Ensemble and generate highly-optimised code using synthesised properties.

2.5.2 FoxNet

FoxNet [82, 83] is a user-space implementation of the TCP/IP protocols in the Standard ML [84] functional programming language. FoxNet was developed as part of the Fox project, which examined the benefits of using high-level programming languages for systems programming. FoxNet comprises a number of modular protocol components. It is possible to reuse these components to build alternative protocol stacks. For example, the authors have also built an implementation of the Domain Name Server protocol using FoxNet. The authors show that functional language techniques can be applied to build composable protocols whilst still maintaining an acceptable level of performance.

2.5.3 Prolac

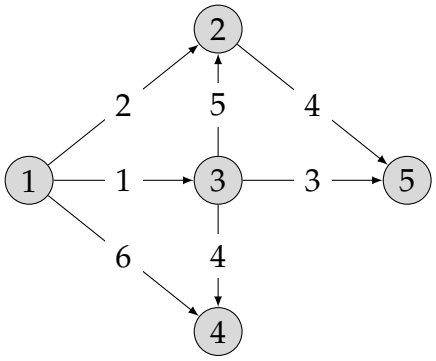
Prolac [85] shares similar goals to the FoxNet project, although instead uses the Prolac DSL for the implementation of networking protocols. Prolac is a statically-typed, object-oriented language that may be compiled to C code. The authors demonstrate how to write a modular TCP in Prolac which can then be run in kernel-space. The main benefit of using Prolac is that the high-level features of the languages allow a clear implementation of the protocol. Furthermore, due to aggressive optimisations within the Prolac compiler, the performance of the Prolac TCP is comparable to that of a hand-coded C implementation.

2.5.4 Melange

Melange [86] is a system for creating network protocols and applications. It comprises two components: a DSL called MPL (the MetaPacket Language) for the specification of packet formats, and an OCaml framework for the implementation of application logic. MPL is itself compiled into imperative-style OCaml code. The authors demonstrate that by combining a packet specification language with the type-safe, garbage collected OCaml language, it is possible to generate high-performance network services such as a DNS or a Secure Shell (SSH) server.

2.6 Algebraic routing

In this section we introduce *routing algebras*. These form the theoretical basis upon which we construct our routing metalanguage. We commence with some basic definitions of graphs before moving onto semiring routing and related generalisations.



(a) Labelled graph

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & 2 & 1 & 6 & \infty \\ \infty & \infty & \infty & \infty & 4 \\ \infty & 5 & \infty & 4 & 3 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \end{array}$$

(b) Adjacency matrix

Figure 2.2: Graphical depiction of example labelled graph $G = (V, E)$ and associated adjacency matrix $\mathbf{A}$. The graph has vertex set $V = \{1, 2, 3, 4, 5\}$ and edge set $E = \{(1, 2), (1, 3), (1, 4), (2, 5), (3, 2), (3, 4), (3, 5)\}$. The arc weights are from the set $\mathbb{N}^\infty$.

2.6.1 Weighted graphs

Suppose that we have a graph $G = (V, E)$, where V corresponds to the set of vertices and $E \subseteq V \times V$ corresponds to the set of edges. We shall require that V and E are finite. We can label a graph by using a *weight function* $w \in E \to S$ for some set S. Given a labelled graph, we can equivalently represent it as an *adjacency matrix*

$$\mathbf{A}(i, j) = \begin{cases} w(i, j) & (i, j) \in E \\ \omega_\otimes & \text{otherwise} \end{cases}$$

where $\omega_\otimes$ is some distinguished element in S (we later place certain requirements on $\omega_\otimes$). Entry $\mathbf{A}(i, j)$ then denotes the weight of arc (i, j), or $\omega_\otimes$ if there is no such arc. Figure 2.2 illustrates an example labelled graph and its associated adjacency matrix. We note that this is a *directed* graph; the presence of an arc (i, j) does not necessarily imply that there is also an arc (j, i). The arc labels are drawn from the set $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ i.e. the set of natural numbers, extended to include the element ∞. Here we take $\omega_\otimes = \infty$.

2.6.2 Paths

Let a non-empty path over G comprise a finite, ordered list of nodes $\mu = \langle i_0, i_1, \ldots, i_m \rangle$, where $(u_k, u_{k+1}) \in E$ for $0 \leq k < m$. Denote the first and the last vertices of a path μ by $\text{first}(\mu) = i_0$ and $\text{last}(\mu) = i_m$. Let ϵ denote the empty path $\langle\rangle$; we shall interpret this path as 'invalid'. Denote the set of all finite paths over G by $\text{paths}(G)$, and the set of finite paths over G from $i \in V$ to $j \in V$ by

$$\text{paths}(G, i, j) = \{\mu \in \text{paths}(G) \mid \text{first}(\mu) = i \wedge \text{last}(\mu) = j\}.$$

Let $\mu, \nu \in \text{paths}(G) \setminus \{\epsilon\}$, with $\mu = \langle i_0, i_1, \ldots, i_m \rangle$ and $\nu = \langle j_0, j_1, \ldots, j_n \rangle$. Define the *path composition* of μ and ν as

$$\mu \hat{\circ} \nu = \begin{cases} \langle i_0, \ldots, i_m, j_1, \ldots j_n \rangle & \text{last}(\mu) = \text{first}(\nu) \\ \epsilon & \text{otherwise.} \end{cases}$$

For $\mu, \nu \in \text{paths}(G)$, extend path composition to operate over the empty path as

$$\mu \circ \nu = \begin{cases} \epsilon & \nu = \epsilon \vee \mu = \epsilon \\ \mu \hat{\circ} \nu & \text{otherwise.} \end{cases}$$

2.6.3 Minimum path weights

In this section we algebraically define the *minimum path weight* between pairs of nodes in a weighted graph. We show how basic, high-level requirements, such as that the minimum path weight is invariant under the order of weight summarisation, naturally lead to the adoption of an algebraic structure known as a *semiring*. We note that more detailed discussion of this topic may be found in [4, 87, 88].

Weight concatenation

Suppose we have a graph $G = (V, E)$, labelled according to the weight function $w \in E \to S$. How might we weight *paths* in G with elements of S? The simplest method is to use a *monoid* $(S, \otimes)$ [89]. We term this the multiplicative monoid (we shortly define a second monoid, called the additive monoid). A monoid has an associative binary operator $\otimes \in S \times S \to S$ and an *identity* $\alpha_\otimes \in S$ satisfying $\alpha_\otimes \otimes s = s \otimes \alpha_\otimes = s$ for $s \in S$. As an aside, if we remove the requirement for an identity then we obtain a semigroup. For our purposes, we additionally require that our monoid has *annihilator* $\omega_\otimes \in S$ satisfying $\omega_\otimes \otimes s = s \otimes \omega_\otimes = \omega_\otimes$ for all $s \in S$.

Paths are weighted by using the multiplicative monoid to *concatenate* the weights of individual arcs. That is, given a path $\mu = \langle i_0, i_1, \ldots, i_m \rangle \in \text{paths}(G) \setminus \{\epsilon\}$, its weight is defined as

$$w(\mu) = w(i_0, i_1) \otimes \cdots \otimes w(i_{m-1}, i_m).$$

Let the weight of the empty path be defined as

$$w(\epsilon) = \omega_\otimes.$$

We illustrate the concatenation of arc weights in Figure 2.3. Suppose that we have some path $\mu = \langle i, \ldots, j \rangle$ with weight $w(\mu) = s$. Now consider extending the path to $\nu = \mu \circ \langle j, k \rangle = \langle i, \ldots, j, k \rangle$ with $w(j, k) = t$. Then, from the definition of the weight function, we have $w(\nu) = w(\mu) \otimes w(j, k) = s \otimes t$.

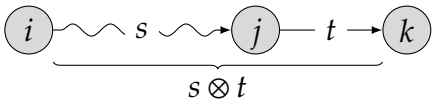


Figure 2.3: Example of concatenating arc weights. Let $\mu = \langle i, \ldots, j \rangle$ with $w(\mu) = s$. Then for the extended path $\nu = \mu \circ \langle j, k \rangle = \langle i, \ldots, j, k \rangle$ with $w(j, k) = t$, we have $w(\nu) = w(\mu) \otimes w(j, k) = s \otimes t$.

We now examine the property requirements. Firstly, associativity implies that given some path $\mu = \langle i, j, k, l \rangle$ then

$$\begin{aligned} w(\mu) &= w(i, j) \otimes [w(j, k) \otimes w(k, l)] \\ &= [w(i, j) \otimes w(j, k)] \otimes w(k, l). \end{aligned}$$

Therefore, it is only necessary to preserve the sequence and not the order of multiplication. This is why the multiplications are not ordered in the definition of the weight function. Secondly, the identity $\alpha_\otimes$ provides a convenient 'initialising value' when incrementally computing the weight of a path; multiplying the weight of a path by $\alpha_\otimes$ has no effect. Finally, should the infinity $\omega_\otimes$ ever be obtained during the incremental computation of a path weight, then the weight of the whole path will also be $\omega_\otimes$. This corresponds to the intuition that it should not be possible to extend the invalid path to obtain a valid path.

One common example of a monoid is Plus $= (\mathbb{N}^\infty, +)$. That is, addition over the natural numbers extended to include infinity. For $n \in \mathbb{N}^\infty$ we extend the usual addition operation to operate over infinity as

$$\infty + n = n + \infty = \infty.$$

Addition is clearly associative and commutative. It also immediately follows from our definition of infinity that $\omega_+ = \infty$. We also have that $\alpha_+ = 0$, because for $x \in \mathbb{N}^\infty$ we have $0 + x = x + 0 = x$. Therefore Plus is indeed a monoid.

Weight summarisation

We now turn to the issue of weight *summarisation*. Suppose that there are two paths $\mu, \nu \in \text{paths}(G, i, j)$ for $i, j \in V$, with $w(\mu) = s$ and $w(\nu) = t$. How might we summarise their weights? Here we use a *commutative* monoid $(S, \oplus)$. We term this the additive monoid. A monoid is commutative if for all $s, t \in S$ we have $s \oplus t = t \oplus s$. For our purposes, we additionally require that $\alpha_\oplus = \omega_\otimes$. This is an example of a *linking axiom* connecting the properties of the additive and multiplicative monoids.

We use the additive monoid to summarise the weights of μ and ν as $w(\mu) \oplus w(\nu) = s \oplus t$. This is demonstrated in Figure 2.4. The commutativity and associativity of $(S, \oplus)$ ensure

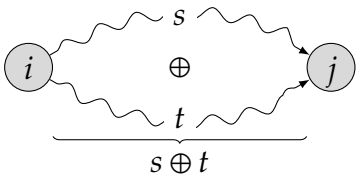
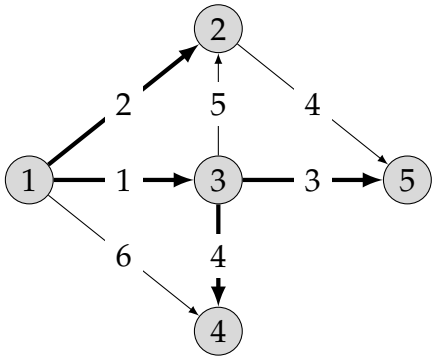


Figure 2.4: Example of summarising path weights. Suppose that we have two paths $\mu = \langle i, \ldots, j \rangle$ and $\nu = \langle i, \ldots, j \rangle$, with $w(\mu) = s$ and $w(\nu) = t$. Then the weights of μ and ν are summarised as $w(\mu) \oplus w(\nu) = s \oplus t$.



(a) Labelled graph

$$\mathbf{A}^* = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 5 & 4 \\ \infty & 0 & \infty & \infty & 4 \\ \infty & 5 & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b) Minimum path weights

Figure 2.5: Example graph and matrix of minimum path weights. The bold arcs depict the minimum cost paths from vertex 1.

that given a set of paths, their weights can be summarised in any order. Furthermore, the requirement that the additive identity is the multiplicative infinity ensures that any path is preferred to the invalid path i.e. for all $\mu \in \text{paths}(G, i, j)$ we have

$$w(\mu) \oplus w(\epsilon) = w(\epsilon) \oplus w(\mu) = w(\mu).$$

A second example of a monoid is minimisation over the natural numbers, extended to include infinity. That is, Min = $(\mathbb{N}^\infty, \min)$, where for $n \in \mathbb{N}^\infty$ we extend the usual minimisation operator to operate over infinity as

$$\min(\infty, n) = \min(n, \infty) = n.$$

Minimisation is both associative and commutative. We also have that $\alpha_{\min} = \infty$ and $\omega_{\min} = 0$. Therefore Min is indeed a monoid.

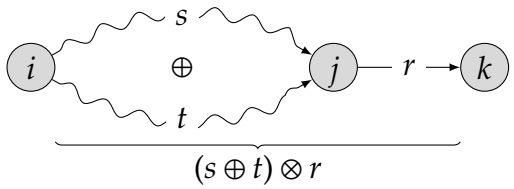


Figure 2.6: Example of using distributivity. Let $\mu = \langle i, \ldots, j \rangle$ and $\nu = \langle i, \ldots, j \rangle$, with $w(\mu) = s$ and $w(\nu) = t$. Then in order to summarise the weights of the two paths $\mu' = \langle i, \ldots, j, k \rangle$ and $\nu' = \langle i, \ldots, j, k \rangle$, we can distribute the common multiplication i.e. $w(\mu') \oplus w(\nu') = (w(\mu) \oplus w(\nu)) \otimes w(j, k) = (s \oplus t) \otimes r$.

Minimum path weights

We now define the *minimum path weight* from i to j as the summarised weight of all paths from i to j i.e.

$$\delta(i, j) = \begin{cases} \alpha_{\oplus} & i = j \\ \displaystyle\sum_{\mu \in \text{paths}(G, i, j)}^{\oplus} w(\mu) & \text{otherwise.} \end{cases} \tag{2.1}$$

Figure 2.5 illustrates the results of computing minimum path weights using our example monoids $(S, \oplus) = \text{Min}$ and $(S, \otimes) = \text{Plus}$. If arc weights are taken to be distance, then this particular algebraic structure yields 'shortest' paths.

2.6.4 Semirings

There are many algorithms for efficiently computing $\delta(u, v)$ instead of individually calculating and then summarising the weights of all paths from u to v; we list several in the next section. One property that may be required by these algorithms is *distributivity*. Given monoids $(S, \oplus)$ and $(S, \otimes)$, we say that the multiplicative operator (right) distributes over the additive operator if for all $s, t, r \in S$, we have

$$(s \oplus t) \otimes r = (s \otimes r) \oplus (t \otimes r).$$

There is also a similar left distribution rule:

$$r \otimes (s \oplus t) = (r \otimes s) \oplus (r \otimes t).$$

We say that a pair of monoids $(S, \oplus)$ and $(S, \otimes)$ are distributive if they are both left distributive and right distributive.

We can now define a semiring. Suppose that we have a pair of monoids $(S, \oplus)$ and $(S, \otimes)$, with the former commutative. Furthermore, suppose that we have the linking axioms (i) $\alpha_{\oplus} = \omega_{\otimes}$, and (ii) $\otimes$ distributive over $\oplus$. Then we call this structure a

semiring. We often denote semirings by the triple $(S, \oplus, \otimes)$. An example of a semiring is MinPlus = $(\mathbb{N}^\infty, \min, +)$.

In Figure 2.10 we summarise the algebraic properties of semirings and related algebraic structures. In this figure, we define properties P and write $\textsc{p}(S)$ to indicate that P holds of the algebraic structure S. For example, we have the property ASSOC, which corresponds to associativity of binary operators. We then write $\textsc{assoc}(S, \oplus)$ to say that $(S, \oplus)$ is associative.

Figure 2.6 demonstrates the significance of distributivity. Suppose that we have two distinct paths, $\mu = \langle i, \ldots, j \rangle$ and $\nu = \langle i, \ldots, j \rangle$, with $w(\mu) = s$ and $w(\nu) = t$. Furthermore, suppose that there is an additional arc (j, k) with $w(j, k) = r$. Extend μ and ν with this arc to obtain the paths $\mu' = \langle i, \ldots, j, k \rangle$ and $\nu' = \langle i, \ldots, j, k \rangle$. Then, using Equation 2.1, we can compute the minimum path weight from i to k as

$$\begin{aligned} \delta(i, k) &= \bigoplus_{\mu \in \text{paths}(G, i, k)} w(\mu) \\ &= w(\mu') \oplus w(\nu') \\ &= (s \otimes r) \oplus (t \otimes r). \end{aligned}$$

However, using the distributivity property, we can effectively 'factorise' the weights of the sub-paths μ and ν:

$$\begin{aligned} \delta(i, k) &= \bigoplus_{\mu \in \text{paths}(G, i, k)} w(\mu) \\ &= w(\mu') \oplus w(\nu') \\ &= (w(\mu) \otimes w(j, k)) \oplus (w(\nu) \otimes w(j, k)) \\ &= (w(\mu) \oplus w(\nu)) \otimes w(j, k) \\ &= (s \oplus t) \otimes r. \end{aligned}$$

Exploiting distributivity can lead to more efficient methods of computing minimum path weights. Furthermore, without distributivity, some algorithms may compute only *locally-minimal* path weights. That is, path weights that are minimal according to the path weights of neighbours, but which are not globally minimal. We further describe the implications of non-distributive semirings in Section 2.6.7.

2.6.5 Matrix semirings

Suppose that we have a seimring $S = (S, \oplus, \otimes)$. Let $M_V(S)$ denote the set of $V \times V$ matrices containing elements from S. Then, somewhat overloading notation, define the *matrix semiring over* S as $(M_V(S), \oplus, \otimes)$, where for $\mathbf{A}, \mathbf{B} \in M_V(S)$ and $i, j \in V$ we define the operators

$$(\mathbf{A} \oplus \mathbf{B})(i, j) = \mathbf{A}(i, j) \oplus \mathbf{B}(i, j).$$

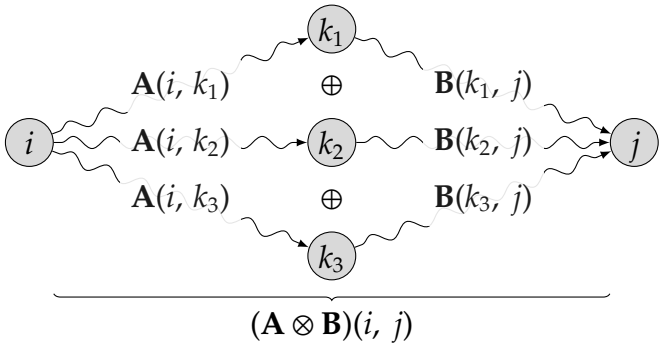


Figure 2.7: Matrix multiplication operation. Suppose that $\mathbf{A}(i)(k)$ and $\mathbf{B}(k)(j)$ correspond to path weights from i to k and k to j, for all $k \in V$. Then $(\mathbf{A} \otimes \mathbf{B})(i)(j)$ is the minimal weight path from i to j using those underlying paths.

and

$$(\mathbf{A} \otimes \mathbf{B})(i,\ j) = \sum_{k}^{\oplus} \mathbf{A}(i,\ k) \otimes \mathbf{B}(k,\ j).$$

The matrix semiring has additive identity

$$\mathbf{W}(i,\ j) = \alpha_{\oplus}$$

and multiplicative identity

$$\mathbf{I}(i,\ j) = \begin{cases} \alpha_{\otimes} & i = j \\ \alpha_{\oplus} & otherwise \end{cases}$$

It is easy to show that in order for $M_V(S)$ to be a semiring, it is only necessary for S to be a semiring i.e. this construction can be applied to *any* semiring to yield a matrix semiring.

Notice that the matrix multiplication operation is defined in terms of both the underlying multiplication and addition operations. Figure 2.7 gives some insight into the behaviour of the matrix multiplication operation. Suppose that $\mathbf{A}(i,\ k)$ and $\mathbf{B}(k,\ j)$ correspond to path weights from i to k and k to j, for all $k \in V$. Then $(\mathbf{A} \otimes \mathbf{B})(i,\ j)$ is the minimal weight path from i to j using those underlying paths.

2.6.6 Routing solutions

In this section we show how to characterise minimal path weights as solutions to equations over matrix semirings. Let $S = (S,\ \oplus,\ \otimes)$ be a (matrix) semiring. For $\mathbf{A} \in S$, define the powers

$$\begin{aligned} \mathbf{A}^0 &= \mathbf{I} \\ \mathbf{A}^k &= \mathbf{A} \otimes \mathbf{A}^{k-1} \quad k > 0 \end{aligned}$$

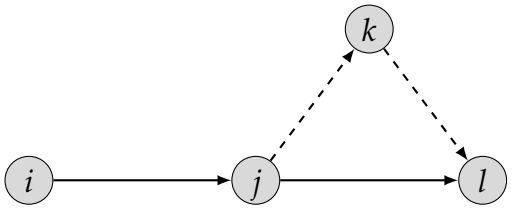


Figure 2.8: Example of locally-optimal path due to non-distributive semiring. The dashed subpath $\langle j, k, l \rangle$ is less preferred than $\langle j, l \rangle$, and is therefore not advertised by j to i. However, in a non-distributive semiring the 'upper' path $\langle i, j, k, l \rangle$ may in fact be more preferred than the 'lower' path $\langle i, j, l \rangle$, causing i to select a path that is only locally-optimal.

Then, define the *closure* of $\mathbf{A}$ as

$$\mathbf{A}^* = \mathbf{A}^0 \oplus \mathbf{A}^1 \oplus \cdots$$

There are several sufficient conditions for the existence of this value, including that $\mathbf{I}$ is the additive infinity [4]. It is easy to show that

$$\mathbf{A}^*(i, j) = \delta(i, j)$$

We also have that $\mathbf{A}^*$ solves for $\mathbf{X}$ the equation

$$\mathbf{X} = (\mathbf{A} \otimes \mathbf{X}) \oplus \mathbf{I}.$$

We term this the *routing* equation. We shall revisit it when discussing the difference between routing and forwarding in Chapter 9.

2.6.7 Bisemigroups

In some cases a structure $S = (S, \oplus, \otimes)$ satisfies all of the semiring axioms with the exception of the distributivity rules. These structures may still be used for finding solutions to the routing equation

$$\mathbf{X} = (\mathbf{A} \otimes \mathbf{X}) \oplus \mathbf{I}.$$

However, solutions $\mathbf{X}$ need *not* satisfy the global optimality condition $\mathbf{X}(i, j) = \delta(i, j)$. They are instead only *locally-optimal* solutions [90, 91]. That is, each element $\mathbf{X}(i, j)$ is optimal given the values $\mathbf{X}(k, j)$ of adjacent nodes $k \in V$. This situation is illustrated in Figure 2.8.

We are interested in non-distributive semirings because they may still be used for representing routing languages, albeit with weaker optimality conditions. We note that BGP is itself non-distributive [90]; operators may be willing to lose global optimality for greater policy control. We define a *bisemigroup* as a semiring that need not satisfy the distributivity axioms.

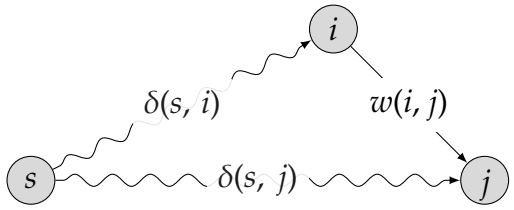


Figure 2.9: Illustration of the relaxation step. The triangle inequality implies that $d[j] \leq d[i] \otimes w(i, j)$. Therefore we set $d[j] = d[j] \oplus (d[i] \otimes w(i, j))$.

2.7 Algorithms

In this section we describe a number of generalised algorithms for computing minimum path weights over graphs labelled with semirings. The metarouting system contains implementations of these algorithms. We present the algorithms in pseudo-code, somewhat akin to C.

2.7.1 Bellman-Ford algorithm

We first describe a generalised version of the Bellman-Ford algorithm [92, 93]. This is a *single source* algorithm. That is, it finds the minimum path weight from some *source* $s \in V$ to each *destination* $d \in V$. The algorithm is illustrated below in pseudo-code:

```
1  BellmanFord(graph G, vertex s)
2    # Initialise
3    for i in vertices(G)
4      d[i] ← α⊕
5    d[s] ← α⊗
6    # Main loop
7    for 1 to size(vertices(G)) − 1
8      for (i, j) in edges(G)
9        alt ← d[i] ⊗ w(i, j)
10       d[j] ← d[j] ⊕ alt
```

The algorithm maintains a vector d[i] containing the current best-estimates of path weights from s to $i \in V$. Initially, in lines 4 – 5, these estimates are set to $\alpha_\oplus$ for $i \in V \setminus \{s\}$ so that any other distance is preferable during the relaxation step (see below). For s, an 'estimate' of $\alpha_\otimes$ is adopted (line 5). This value acts as the identity for subsequent path weight multiplications.

The main body of the algorithm occurs in lines 7 to 10. Here, each edge is 'relaxed' $|V| - 1$ times. That is, given an edge $(i, j) \in E$, the current best-estimate path-weight to j is compared with the path weight formed by combining the minimum path weights

from s to i and from i to j. The more-preferred of these two values is then adopted. In this manner, minimum path weights are propagated along paths. This process is called relaxation because it ensures that the triangle inequality is satisfied. The triangle inequality, illustrated in Figure 2.9, states that the minimum path weight from s to j should always be more preferred than the combined minimum path weights from s to i and from i to j. When this algorithm terminates, we have that $\mathrm{d[i]} = \delta(s, i)$.

It is straightforward to derive the time complexity of this algorithm. In lines 7 to 10, each edge is relaxed $|V| - 1$ times, and there are $|E|$ such edges. Therefore the time complexity of the generalised Bellman-Ford algorithm is $O(|V|.|E|)$. In the worst case, $|E| = |V|^2$, and the time complexity becomes $O(|V|^3)$.

The Bellman-Ford algorithm has relatively weak property requirements for convergence, only requiring that metrics become less preferred when policy is applied. A distributed version of the algorithm is found in vectoring routing protocols such as RIP [13] and BGP [9]. A significant problem with the algorithm is that unless paths are explicitly tracked (as is the case with BGP), then 'counting to infinity' can occur. This happens when a route become unavailable, but old information about it still persists in the network. It can cause network-wide cycles, where each router believes that its neighbour still has a valid route to the destination. The result is that metric values continually increase until the top most metric (infinity) is reached, at which point the route is removed.

2.7.2 Dijkstra's algorithm

The next algorithm is a generalised version of Dijkstra's algorithm [10]. Again, this is a single source algorithm. In contrast to the Bellman-Ford algorithm, Dijkstra's algorithm requires a stronger set of algebraic properties. In particular, it is necessary for $(S, \oplus, \otimes)$ to be a semiring, and for $\oplus$ to be idempotent and selective. The generalised algorithm is shown below in pseudo-code:

```
1  Dijkstra(graph G, vertex s)
2    # Initialise
3    for i in vertices(G)
4      d[i] ← α⊕
5    d[s] ← α⊗
6    Qd ← vertices(G)
7    # Main loop
8    while (not empty(Qd))
9      i ← extract_head(Qd)
10     for j in adj(i)
11       alt ← d[i] ⊗ w(i, j)
```

12 $\quad d[j] \leftarrow d[j] \oplus alt$

The algorithm maintains a vector d[i] containing the current best-estimates of path weights from s to $i \in V$. Initially, these estimates are set to $\alpha_\oplus$ for $i \in V \setminus \{s\}$ (line 4), so that any other path weight is preferred during the relaxation step (see below). For s, the estimate is set to the initialising value of $\alpha_\otimes$ (line 5). The algorithm also has a queue of vertices Q_d, ordered by increasing path weight estimates d[i]. Initially, all vertices are placed into the queue (line 6). At each iteration of the algorithm (lines 8 to 12), the vertex i at the head of the queue is extracted. At this point d[i] corresponds to the minimum path weight from s to i. Finally, each neighbour j of i is relaxed (lines 10 to 12). At termination, we have that $d[i] = \delta(s, i)$.

There has been extensive work in reducing the time and space complexity of this algorithm. Large improvements can be made purely by adopting more efficient queue implementations. We briefly summarise the basic results from [93]. The simplest implementation of Dijkstra's algorithm uses an array. This requires $O(|V|)$ comparisons to find the minimum element. Each vertex is extracted once, leading to V such operations, and hence an overall time complexity of $O(|V|^2)$ The running time can be improved if the graph is *sparse* i.e. there are relatively few edges. In particular, if $E = o(|V|^2 / \log|V|)$ then the queue can be implemented using a binary min-heap. Each update and queue extraction costs $O(\log|V|)$. There are at most $|E|$ of the former, and $|V|$ of the latter, resulting in a running time of $O((|E|+|V|)\log|V|)$. This bound can be improved still further by adopting a Fibonacci heap. The queue extraction remains at $O(\log|V|)$, whilst the update cost is reduced to $O(1)$. This results in time complexity of $O(|V|\log|V| + |E|)$.

Under certain assumptions, the running time of Dijkstra's algorithm can be improved still further. One approach involves *scaling*, whereby the edge weights are modified so that more efficient, non-numeric algorithms may be used. Using this technique, Gabow [94] showed how to obtain a $O(|V|^{\frac{3}{4}}|E|\log N)$ running time, where N is the largest edge weight or path weight estimate. More recently, Thorup [95] has shown how to find the single-source shortest path in linear time and space on a RAM model using a restricted set of instructions. This approach assumes that the graph is undirected and that the weights are bounded, positive integers.

Dijkstra's algorithm is used in both the OSPF [11] and IS-IS [12] routing protocols. Both of these protocols use a link-state flooding mechanism to propagate reachability information between routers (in the same 'area'). Each router then individually computes shortest-paths using Dijkstra's algorithm. In the case of multiple areas, a distance vectoring mechanism based upon the Bellman-Ford algorithm is additionally used.

2.7.3 Iterative matrix algorithm

We now give an algorithm for finding minimal path weights between *all pairs* of nodes [5]. We term this algorithm the *iterative matrix algorithm*. It can be viewed as a version of the Bellman-Ford algorithm that has been generalised to compute single-source minimal path weights for all sources in parallel.

For a matrix semiring $S = (S, \oplus, \otimes)$, with $\mathbf{A} \in S$, define the algorithm as

$$\begin{aligned} \mathbf{A}^{(0)} &= \mathbf{I} \\ \mathbf{A}^{(k+1)} &= \mathbf{A}^{(k)} \oplus \mathbf{A}^{k+1} \quad k \geq 0 \end{aligned}$$

For matrix size n and k iterations, this algorithm has time complexity of $O(k.n^3)$. Under certain conditions this algorithm converges in n iterations, leading to a time complexity of $O(n^4)$.

2.7.4 Recursive matrix algorithm

A variant of the iterative matrix algorithm is the *recursive* matrix algorithm. This algorithm is defined as follows:

$$\begin{aligned} \mathbf{A}^{[0]} &= \mathbf{I} \\ \mathbf{A}^{[k+1]} &= (\mathbf{A}^{[k]} \otimes \mathbf{A}) \oplus \mathbf{I} \quad k \geq 0 \end{aligned}$$

It is easy to show that if distribution holds, then $\mathbf{A}^{[k]} = \mathbf{A}^{(k)}$. We give the proof below.

Proof. Proceed by induction. Base case holds by definition. Inductive case:

$$\begin{aligned} \mathbf{A}^{[k+1]} &= \mathbf{A}^{[k]} \otimes \mathbf{A} \oplus \mathbf{I} \\ &= \mathbf{A}^{(k)} \otimes \mathbf{A} \oplus \mathbf{I} && \text{(induction hypothesis)} \\ &= (\bigoplus_{0 \leq l \leq k} \mathbf{A}^l) \otimes \mathbf{A} \oplus \mathbf{I} \\ &= (\bigoplus_{1 \leq l \leq k+1} \mathbf{A}^l) \oplus \mathbf{I} && \text{(left distribution)} \\ &= \bigoplus_{0 \leq l \leq k+1} \mathbf{A}^l \\ &= \mathbf{A}^{(k+1)} \end{aligned}$$

□

Property name	Definition	Semigroup	Monoid	Semiring $\oplus$	Semiring $\otimes$
ASSOC	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	✓	✓	✓	✓
COMM	$x \oplus y = y \oplus x$			✓	
ALPHA	$\exists\, \alpha_\oplus \in S.\, x \oplus \alpha_\oplus = \alpha_\oplus \oplus x = x$		✓	✓	✓
OMEGA	$\exists\, \omega_\oplus \in S.\, x \oplus \omega_\oplus = \omega_\oplus \oplus x = \omega_\oplus$				✓
IDEM	$x \oplus x = x$				
SEL	$x \oplus y \in \{x, y\}$				
LEFT_CANC	$x \oplus y = x \oplus z \Rightarrow y = z$				
RIGHT_CANC	$y \oplus x = z \oplus x \Rightarrow y = z$				
LEFT_CONST	$x \oplus y = x \oplus z$				
RIGHT_CONST	$y \oplus x = z \oplus x$				

(a) Properties of semigroups $(S, \oplus)$

Property name	Definition	Bisemigroup	Semiring
LEFT_DIST	$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$		✓
RIGHT_DIST	$(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$		✓
ALPHA_OMEGA	$\text{ALPHA}(S, \oplus) \wedge \text{OMEGA}(S, \otimes) \wedge \alpha_\oplus = \omega_\otimes$		✓
OMEGA_ALPHA	$\text{OMEGA}(S, \oplus) \wedge \text{ALPHA}(S, \otimes) \wedge \omega_\oplus = \alpha_\otimes$		
LEFT_INCR	$x \oplus y = x \Rightarrow (z \otimes x) \oplus (z \otimes y) = z \otimes x$		
RIGHT_INCR	$x \oplus y = x \Rightarrow (x \otimes z) \oplus (y \otimes z) = x \otimes z$		

(b) Properties of bisemigroups $(S, \oplus, \otimes)$

Figure 2.10: Properties of semigroups, bisemigroups and related structures. Free variables are universally quantified.

Chapter 3

System architecture

In this chapter we describe the high-level architecture of the metarouting system. We commence with an overview of the system design (§ 3.1). We then describe the individual components of the system: the *routing interface* that defines the interactions between routing languages and algorithms (§ 3.2), how routing languages are compiled to satisfy the routing interface (§ 3.3) and the implementation of the routing algorithms (§ 3.4). Finally we discuss the user interfaces to the metarouting system, including how to configure generated protocols (§ 3.5).

3.1 Design overview

A routing protocol can be decomposed into three components: a language, an algorithm and a proof of correctness. In this section we examine each component, and show how this logical separation can be exploited to produce an architecture for the creation of new routing protocols from high-level specifications.

Recall that a routing language abstractly specifies the types of metrics and policy, and the results of applying policy to metrics. We wish to allow users the flexibility to define their own routing languages, instead of being limited to the pre-defined routing languages of current protocols. For this purpose we define a domain-specific language for specifying routing languages i.e. a metalanguage. We term this language the Routing Abstract Metalanguage (RAML). We describe restricted and extended variants of RAML, termed RAML_1 and RAML_2, in Chapters 5 and 7 respectively. In particular, we draw attention to the simple example of a RAML_1 routing language that is given in Section 5.1.

We would like users to be able to specify *what* a routing language should compute without defining *how* it should do this. For example, users should not be concerned with the specifics of choosing the appropriate data-type to represent a set, nor how to efficiently implement the union operation. This leads us to choose a declarative design

for RAML. In Chapter 8 we show how this approach allows the metarouting system the freedom to select efficient datatypes itself.

We convert RAML specifications into fast, executable code by compiling the language into C++. We have chosen to compile RAML, instead of interpreting it, for efficiency reasons; routing is performance-sensitive and an interpreted language implementation is typically slower than a compiled implementation. Furthermore, our particular approach to compilation has a relatively light-weight implementation, and therefore we suffer little from the increase in complexity that is usually associated with compiled language implementations. We give an architectural overview of compilation in Section 3.3 and describe the implementation details in Chapter 6.

The definition of RAML embodies a delicate tension between expressivity and computability. On the one hand, we wish for RAML to be able to express a *useful* set of routing languages. Although it is impossible to formally define this set, it is certainly desirable that we can express those routing languages that are embodied in current routing protocols. On the other hand, we would like to be able to automatically prove correctness of the resulting protocols. If the metalanguage is too expressive then it is possible that we might no longer be able to automatically prove correctness for all definable routing languages. Our current version of RAML is sufficiently constrained so as to maintain decidability of correctness. We believe that it is still possible to define a wide range of useful routing languages within RAML. We discuss the semantics of RAML, including automatic property inference, in Chapter 4.

Now suppose that we have an executable routing language, perhaps compiled from a RAML specification. In order to define a routing protocol, it is necessary to combine the routing language with a routing algorithm. Note that we can define a routing algorithm independently of a particular choice of routing language. For example, Dijkstra's algorithm (§ 2.7.2) requires only that a routing language contains the operations $\oplus$ and $\otimes$ over some set S, and the constants $\alpha_{\oplus}$, $\alpha_{\otimes} \in S$ (in addition to some algebraic properties). This suggests that we can create a *library* of pre-defined, generic routing algorithms that can be combined with compiled routing languages to produce routing protocols. We further describe routing algorithms in Section 3.4.

In practice, useful routing algorithms require that routing languages implement additional functions. For example, in order to configure a routing protocol, we need some way of converting textual representations of metrics and policy into elements of S. Similarly, to debug a routing protocol it is necessary to convert elements of S back into textual representations. Distributed algorithms also require some way of converting elements of S into efficient byte-strings for transfer over the network. This approach requires that there is a well-defined interface between compiled routing languages and the generic routing algorithms. We term this a *routing interface*. We describe the routing interface in Section 3.2.

To summarise, our system comprises (i) a declarative language, called RAML, for the specification of routing languages, (ii) a compiler to generate efficient, executable code from RAML specifications, (iii) a library of generic routing algorithms that can be combined with compiled routing languages and (iv) a routing interface that defines how routing languages and routing algorithms interact.

3.2 Routing interfaces

A routing interface specifies the permitted interactions between a routing language and a routing algorithm. In our case, we specify a routing interface as a collection of C function prototypes; a compiled routing algorithm then contains code to implement these functions, whilst a routing algorithm may only access these functions within the routing language. Figure 3.1 gives an example interface for a vectoring algorithm such as the distributed Bellman-Ford algorithm employed by RIP. In the remainder of this section, we describe this interface and explain the reasons for adopting this particular level of abstraction.

The `init` and `deinit` functions are required to respectively initialise and deinitialise any state in the routing language implementation. Examples of such state include persistent data structures such as hash-tables and queues. We require that a routing algorithm executes the `init` function exactly once and before any other the other routing interface functions are invoked. Similarly, a routing algorithm must invoke the `deinit` function exactly once, and no other routing interface functions may be subsequently invoked.

The `id` function returns a hashed representation of the routing language. Hashes are computed at compile time by applying a hash function to the abstract syntax tree of routing languages. The purpose of hashes is to detect configuration errors whereby adjacent routing daemons communicate using different routing languages. These situations are undesirable because we are not able to guarantee that such 'hybrid' protocols are well-behaved (allowing safe interaction between different routing languages remains an area for future work; such a facility might be useful when upgrading deployed routing protocols, for example). Routing daemons use hashes to detect such errors by checking that adjacent daemons have identical routing language hashes to their own. We note that this is only a probabilistic guarantee because hash collisions are possible, although the likelihood of such errors can be reduced by using larger hash sizes.

Metrics and policy are represented within the interface as abstract values of type `void *`. By design, routing algorithms cannot access such values directly and instead must indirect via the routing interface. Furthermore, values of these types must be explicitly copied and freed, using the `copy` and `free` functions respectively. This ensures that

the routing language implementation has maximum flexibility when choosing concrete implementations of these types. For example, integers may have literal representations, whilst more complex data-structures may be heap-allocated.

The `metric_hash` function computes the hash of a metric. This facility is used by some routing algorithms to implement a limited form of sharing. This prevents equivalent values from being represented more than once in the runtime. In Chapter 8 we show how to systematically enable sharing within compiled routing languages, thus largely eliminating the need for this function. Note that we have been able to introduce such optimisations without altering the routing interface or any of the routing algorithms. This supports the claim that the routing interfaces introduce an appropriate level of abstraction.

A routing algorithm is typically configured using textual representations of metrics and policy. The routing language must therefore provide some mechanism of parsing such values into the internal representations. For this purpose there is the `metric_parse` function which converts a character buffer into a `metric_t`. Similarly, in order for an operator to understand the behaviour of a routing protocol, it must be possible to obtain a textual representation of a metric or policy value. This facility is provided by the `metric_print` function, which places a textual representation of a `metric_t` into the specified buffer.

A distributed routing algorithm requires a method of efficiently transferring metrics (and also policy, for Dijkstra-like algorithms) between daemons. For a conventional routing protocol, the 'wire' representation of these values is specified as part of the packet format. For us, the wire representation potentially changes for each routing language. Therefore a routing language must provide `metric_marshal` and `metric_unmarshal` functions. The former converts a `metric_t` value into a linear sequence of bytes so that it may be transferred across the network, whilst the latter performs the inverse operation to obtain a `metric_t`.

A routing algorithm typically requires one or more distinguished constants. In this example, the algorithm requires an 'infinity' ($\alpha_{\oplus}$) metric value to denote invalid routes. This facility is provided by the `metric_infinity` function, which returns the required value. Furthermore, routing algorithms must typically test for equality against such constants. Here, this facility is provided by the `metric_is_infinity` function. The interface also provides the function `metric_is_equal` for testing for equality between any two metrics. Note that although any invocation of `metric_is_infinity` could be replaced by calls to `metric_infinity` and `metric_is_equal`, retaining this function leads to more idiomatic code in routing algorithms.

Up to this point, we have mostly described what are essentially utility functions. We now describe the interface between the compiled algebraic operators. The example interface assumes that summarisation is defined as an order (instead of as a binary oper-

ator; see Chapter 7). The compiled order is exposed as the function `metric_is_better`. This function accepts two `metric_t` values x and y, and returns a boolean value indicating whether x is preferred to y. Policy application is implemented as the function `policy_apply`. This takes a `metric_t` value and a `policy_t` value, and applies the policy to the metric to obtain another `metric_t` value.

3.3 Compilation

Again, recall that a routing protocol comprises a language, an algorithm and a proof of correctness. Our compiler accepts protocol specifications comprising a routing language and a routing algorithm. The routing language component is specified in RAML file, whilst the routing algorithm is specified as a command-line option that names one of several pre-defined algorithms. The compiler automatically checks the language specification for correctness by determining its algebraic properties and verifying that they match those required by the algorithm. The output of the compiler is an executable routing protocol.

The compiler architecture can be logically divided into several phases, summarised in Figure 3.5. Firstly, there is the front-end that performs basic parsing of protocol specifications. Next there is routing language processing, where the routing language component is translated into an intermediate language, termed the Intermediate Routing Language (IRL). We discuss variants of IRL, termed IRL_1 and IRL_2 in Chapters 4 and 7 respectively. After this is the property-checking stage, where the algebraic properties of the IRL term are automatically inferred, and checked against those required by the algorithm. The result is an Annotated Intermediate Routing Language (AIRL) term. Next is the back-end, where AIRLis translated into C++ and then, using a standard C++ compiler, into object code. Finally, the object code is linked against the specified routing algorithm to produce a routing protocol.

One important design choice concerns our use of an intermediate language. This feature is standard within compiler architectures, allowing a clean separation between the front-end and back-end. Providing that the intermediate language is sufficiently expressive, it is often possible to augment the input language by modifying just the front end. This feature has been of considerable benefit to us when we have added features to RAML. A second advantage of an intermediate language is that it is often possible to add additional target languages without modifying the front-end, although we do not exploit this facility because we currently only target C++.

Our intermediate language, IRL, closely matches the underlying algebraic semantics of RAML, containing explicit mathematical objects such as sets and binary operators. This reduces the complexity of the implementation and therefore minimises the scope

```
// Initialise and deinitialise routing language
void init(void);
void deinit(void);

// Identity of routing language
const unsigned char * id(void);

// Type of metrics and policy
typedef void * metric_t;
typedef void * policy_t;

// Explicit memory management for metrics
metric_t metric_copy(metric_t);
void metric_free(metric_t);

// Identity of metrics
uint32_t metric_hash(metric_t);

// Interconversion of metrics
metric_t metric_parse(const char *);
size_t metric_print(char *, size_t, metric_t);
size_t metric_marshal(void *, size_t, metric_t);
metric_t metric_unmarshal(const void *, size_t);

// Constants for metrics
metric_t metric_infinity(void);
int metric_is_infinity(metric_t);

// Comparisons for metrics
int metric_is_better(metric_t, metric_t);
int metric_is_equal(metric_t, metric_t);

// Policy application
metric_t policy_apply(policy_t, metric_t);
```

Figure 3.1: An extended fragment of the C *routing interface* that mediates the interactions between a compiled routing language and a routing algorithm. This particular routing interface is for a distributed, vectoring algorithm. Note that for brevity, functions for parsing, printing, marshalling and copying some types have been omitted.

for implementation errors; for a given RAML term, there is often a close resemblance between the 'on-paper' semantics and the IRL translation. The second advantage of our intermediate language occurs when checking for correctness. Our notions of correctness are given in terms of the properties of the underlying algebraic objects. Therefore we can easily map these correctness conditions onto IRL.

The actual property checking occurs in a 'bottom-up' manner. The algebraic properties of atomic terms are already known, and there are various rules for inferring properties of non-atomic terms from the properties of their subterms. Therefore we synthesise the algebraic properties of a given term by first computing the properties of its subterms. During property checking, we also annotate each IRL term with its computed properties to obtain terms within the Annotated Intermediate Routing Language (AIRL). The reason for maintaining the property information is that it is used by the back-end to generate more efficient target code. Note that the distinction between IRL and AIRLis somewhat artificial because the syntactic structure of AIRLis simply that of IRL, but augmented with algebraic properties.

The compiler targets C++ because the routing algorithms (§ 3.4) are themselves written in C/C++, and therefore it is straightforward to interface with this code. An additional benefit from compiling to C++ is that its template metaprogramming facilities allow a straight-forward embedding of IRL. We also benefit from the fact that C++ is a mature language, benefiting from high-quality optimising compilers. In Chapter 6 we explain the actual process of generating C++ code.

The output of the compiler is dependent upon the type of target algorithm. In brief, there are two types of algorithm: *offline* and *online* (§ 3.4). The former include Dijkstra's algorithm, and are used for computing routing solutions over static topologies, whilst the latter are generalised version of the Quagga routing protocols. In the case of offline algorithms the compiled routing language is statically linked with the algorithm to produce a stand-alone executable. For online algorithms, the compiled routing language is emitted as a shared library. The algorithms load the shared libraries at runtime based upon their configuration. The advantage of this approach, over statically linking the library, is that we can rapidly alter the routing language without having to recompile the routing algorithm.

3.4 Routing algorithms

Our system contains two kinds of routing algorithms: *offline* and *online*. The offline algorithms are executable versions of the algorithms listed in Section 2.7. Offline algorithms essentially capture the pure algorithmic component of routing. The implementation of these algorithms is relatively straightforward, essentially comprising some form of iteration over a matrix structure. We do not further detail the implementation specifics.

In contrast, online algorithms are used for actual network routing. This latter kind of algorithm can be seen as an elaboration of the former, additionally concerned with details such as IP addresses, packet formats and timers. The implementation of online algorithms is therefore vastly more complicated. Fortunately, there are many mature implementations of online routing algorithms in the form of current routing protocols. The difficulty is that each of these algorithms have been specialised to operate upon a particular routing language. Therefore, for each of our chosen algorithms, we have removed the hard-wired routing language and replaced it with calls to a routing interface. Currently we have modified the RIP, BGP and OSPF daemons from the Quagga routing suite [36], although this technique could also be applied to other protocol implementations, such those found in the XORP routing suite [22].

Figure 3.6 shows some of the changes made to the Quagga RIP `rip_response_process` function when generalising it to interface to arbitrary routing languages. This code is responsible for handling incoming routing updates. The excerpted section shows how metrics are unmarshalled and checked for validity. Figure 3.6(a) shows the original code. Here metrics are unmarshalled by converting a 32-bit integer from network to host byte-order using the function `ntohl`. The resulting value is then checked to ensure that it is within the permitted range [1, 16].

Figure 3.6(b) shows the results of the generalising the Quagga RIP `rip_rte_process` function to use a routing interface. Unmarshalling now occurs via the routing interface function `mrc_wire_metric_unmarshal` (this is a slightly more general function than the `metric_unmarshal` presented earlier, although its purpose is essentially the same). The first parameter of this function corresponds to the particular routing language in use, whilst the next two parameters are a pointer to a buffer containing a marshalled metric and the length of that buffer (the size of marshalled metrics can vary, depending upon the routing language). The unmarshalled metric is then tested for validity using the function `mrc_wire_metric_is_null` – unmarshalling returns a value of zero if there is an error.

Figure 3.6 succinctly illustrates how metrics become opaque entities upon generalisation. Whereas in Figure 3.6(a) it is possible to perform an explicit comparison against integer literals, in Figure 3.6(b) such error checking must be performed using the routing interface. This abstraction is necessary because the representation of metrics is dependent upon the particular routing language, and therefore the routing algorithm is not able to assume anything about the particular representation of metrics.

Whilst generalising the Quagga routing daemons, we have mostly performed only *local* changes. That is, most modifications have only affected a few lines at a time, and have simply replaced concrete manipulations of metrics with calls to the appropriate routing interface. Significantly, our modifications have not affected the large-scale structure of the source code. Again, this point is illustrated in Figure 3.6, where we

have performed two local changes: firstly, to generalise the unmarshalling code, and secondly, to generalise the error checking code. This supports the view the separation of routing protocols into a routing language and a routing algorithm is a 'natural' decomposition; the original protocols have already been implemented in such a way so as to make the underlying routing algorithms easily accessible.

Figure 3.2 provides a coarse view of the amount of work required to generalise each routing protocol. The diff size is the number of lines output by the standard Unix `diff` tool when comparing the source trees for the original and generalised routing daemons. This value essentially corresponds to the number of lines of code that have been added, removed or modified. The total size is simply the total number of lines of code (including comments) in the source tree for the generalised routing daemons. Note that all lines count values are given in thousands of lines of code.

The values in Figure 3.2 show that generalising existing protocols to use a routing interface requires comparatively little work especially when compared to the total cost of implementation. The Quagga BGP and RIP daemons require modifications of just 300 and 500 lines of code respectively. The larger number of modifications required by OSPF is caused by the original protocol directly copying each routing update from the wire into a single buffer, and then performing in-place modifications. This design increases the performance of the protocol, but requires fixed-size metrics. Therefore it is necessary to (slightly) more radically alter the OSPF implementation to accommodate variable-sized metrics.

In the RIP and BGP daemons, our modifications have an associated performance penalty caused by the overhead of invoking the routing interface. Whereas an unmodified routing protocol can directly manipulate a metric or policy value, the generalised protocols must now make at least one function call to invoke the compiled routing language. We have very approximately quantified the execution time penalty as being around ten percent of the time taken by an unmodified daemon, although further experiments are required.

We now take a moment to attribute the work presented in this section. The general system design, including the routing interfaces of Section 3.2, is the result of the author's own work. Many of the particular design choices were arrived at by the author performing a prototype generalisation of the XORP RIP routing daemon. The particular generalisations of the Quagga BGP, OSPF and RIP routing daemons that are included in the current metarouting system were performed by Philip Taylor and Md. Abdul Alim. However, the resulting modified daemons adhere to the general system design created by the author.

Protocol	Diff size	Total
BGP	0.3	54
OSPF	2.9	47
RIP	0.5	11
Zebra	0.2	24
Shared	0.3	42

Figure 3.2: Summary of lines of code added or changed in Quagga compared to total number of lines of generalised code. 'Zebra' is the central daemon that co-ordinates each of the routing daemons, and 'shared' is the shared library code used by all protocol implementations. Diff sizes and totals are given in thousands of lines of code.

3.5 User interfaces

In this section we describe the user interfaces to the metarouting system and the generated protocols. We aim to give an intuition for the ways in which a user might interact with the system, rather than an exhaustive reference.

Consider the following RAML routing language that defines a routing language that is based upon the MinPlus algebra of Chapter 2:

```
let min_plus : bisemigroup = min_plus_bound(W, 0, 100)
```

In this language, metrics and policies both take the form of integers in the range 0 to 100 (inclusive), whilst the value `W` denotes an error condition. We describe the metalanguage in more detail in Chapter 5. In this section we show how to compile this routing language into offline and online routing protocols. We first consider the offline case.

3.5.1 Offline routing protocols

Suppose that the `min_plus` routing language specification is contained in a file named `min_plus.aml` (the `.aml` suffix denotes 'abstract metalanguage'). We compile the language to use the iterative matrix algorithm as

```
mrc -matrix min_plus.aml
```

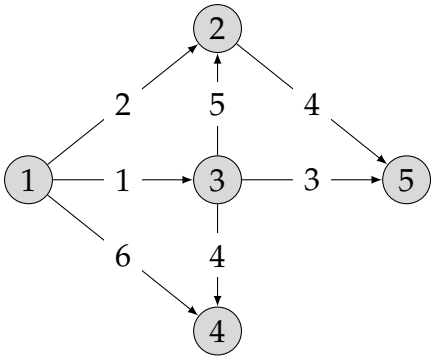
The `mrc` executable is the metarouting compiler. The result of running this command is that an executable named `min_plus` is generated. This is an offline routing protocol.

Offline routing protocols operate upon textual representations of labelled graphs that we term *graph descriptions*. An example of a graph description for the `min_plus` routing

nodes = {
  <**name** = n1, **originate** = 0>,
  <**name** = n2, **originate** = 0>,
  <**name** = n3, **originate** = 0>,
  <**name** = n4, **originate** = 0>,
  <**name** = n5, **originate** = 0>
}

arcs = {
  <**from** = n1, **to** = n2, **policy** = 2>,
  <**from** = n1, **to** = n3, **policy** = 1>,
  <**from** = n1, **to** = n4, **policy** = 6>,
  <**from** = n2, **to** = n5, **policy** = 4>,
  <**from** = n3, **to** = n2, **policy** = 5>,
  <**from** = n3, **to** = n4, **policy** = 4>,
  <**from** = n3, **to** = n5, **policy** = 3>
}

(a) Graph specification



(b) Labelled graph

–>	n1	n2	n3	n4	n5
n1	0	2	1	5	4
n2	W	0	W	W	4
n3	W	5	0	4	3
n4	W	W	W	0	W
n5	W	W	W	W	0

(c) Matrix output

Figure 3.3: Example graph specification and matrix output for the `min_plus` routing language with the matrix algorithm.

language is given in Figure 3.3(a). The format specifies the names of nodes and the weighted arcs connecting these nodes. Each node specification includes an *origination* metric that defines the initial value of metrics originated by that node. The corresponding graphical depiction (without originated values) is given in Figure 3.3(b). This is the same graph that we saw in Chapter 2. As an aside, each offline protocol is also able to *randomly* generate graph specifications, although we do not further discuss this feature.

We shall assume that the graph description of Figure 3.3(a) is located in a file named `min_plus.arc`. The `min_plus` protocol is executed upon the graph description as

```
min_plus min_plus.arc
```

Figure 3.3(c) shows the resulting output. This represents a matrix of values from the routing language. As expected, these values do indeed correspond to the shortest path weights. Note that it is also possible to 'linearise' the output for viewing large graphs.

3.5.2 Online routing protocols

We now consider how to generate online routing protocols. As an example, we shall use the same `min_plus` routing language. Suppose that we wished to use the generalised

```
routing-language ./min_plus.so

route-metric m
  set 0
end-metric

route-policy p in
  set 10
end-policy

router rip
  default-metric m
  default-policy p in
  network eth0
  network eth1
```

Figure 3.4: Example configuration for the `min_plus` routing language with the generalised RIP algorithm

RIP algorithm (essentially, distributed Bellman-Ford). We invoke the metarouting compiler as

```
mrc -grip min_plus.aml
```

The resulting output is a shared library named `min_plus.so`. This library must be loaded into the generalised RIP algorithm before it can be executed. This occurs using a Quagga configuration file.

We give an example of a configuration file in Figure 3.4. We note that the metarouting-specific syntax extensions for the configuration of online routing protocols have been developed by Philip Taylor. We now briefly highlight these extensions. The initial `routing-language` command specifies the location of the shared library containing the routing language implementation. Metrics and policy are defined using the `route-metric` and `route-policy` commands respectively. Here we define a metric `m` of value 0 and a policy `p` of value 10. The `default-metric m` sets `m` to be the metric value of originated routes (analogously to the origination metric in the previous section). The `default-policy p in` sets `p` to be the default policy that is applied when routes are received. Online routing protocols may also be interactively configured using the telnet interface that is exposed by each routing daemon, although we do not explain the details. Each telnet interface also allows the state of the running protocol to be examined in the usual manner.

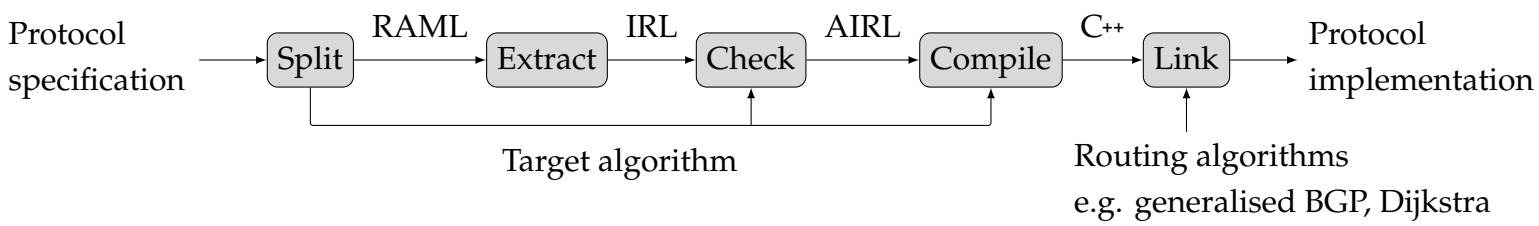


Figure 3.5: Compiler architecture. The boxes and arcs respectively denote compiler stages and the flow of data between stages, with arc labels corresponding to the particular language generated by each stage.

```
/* Convert metric value to host byte order */
rte->metric = ntohl (rte->metric);

/* Is the metric valid (i.e., between 1 and 16)? */
if (! (rte->metric >= 1 && rte->metric <= 16))
{
  zlog_info ("Route's metric is not in the 1-16 range.");
  rip_peer_bad_route (from);
  continue;
}
```

(a) Original code

```
/* Unmarshal metric value */
mrc_wire_metric_free (routing_language, wire_metric);
wire_metric = mrc_wire_metric_unmarshal (
  routing_language, (char *) &rte->metric,
  ntohl (rte->rte_len) - RIP_BASE_RTE_SIZE);

/* Is the metric valid? */
if (mrc_wire_metric_is_null (wire_metric))
{
  zlog_info ("Route's metric is not valid.");
  rip_peer_bad_route (from);
  continue;
}
```

(b) Generalised code

Figure 3.6: Example of generalising the source code for the Quagga RIP daemon (comments edited for brevity, otherwise unchanged). Here we show changes made to the `rip_response_process` function. This code handles incoming route updates and hence must unmarshal route metrics. Note that the modifications are *local*; they do not affect the large-scale program structure.

Chapter 4

Semantic domain

In this chapter we define the semantic domain which we use to assign a meaning, or *semantics*, to our metalanguage. Firstly we define the relationships between the metalanguage, the intermediate language, the target language, and the semantic domain (§ 4.1). Next, we give some basic mathematical definitions (§ 4.2). This serves as a foundation for subsequent sections. We then define the semantic objects, comprising semigroups (§ 4.3) and bisemigroups (§ 4.4). We also give rules which specify the mathematical properties of algebras defined using these constructions. Finally, we define the intermediate language (§ 4.5). This is used for programmatically representing mathematical objects from the semantic domain, and also acts as a bridge between the metalanguage and the target language.

4.1 Overview

Our system contains a number of languages. Firstly, there is the metalanguage, RAML, which is used for defining routing languages. We define restricted and extended versions of the metalanguage in Chapters 5 and 7 respectively. Next, there is the intermediate language, IRL, which contains explicit representations of types, binary operators, orders etc. Metalanguage terms are translated into intermediate language terms by the front-end of the compiler (§ 3.3). Finally, there is the target language, C++, into which the compiler translates terms from the intermediate language. Generated C++ code is then compiled into executable programs using a standard compiler. We illustrate the relationship between these three languages in Figure 4.1. The translation function $(\!|\_|\!)$, which maps from RAML terms onto IRL terms, is defined in Chapter 5.

We formally relate RAML and IRL terms using a semantic domain, D, which we define in this chapter. The semantic domain contains mathematical objects such as sets, semigroups and bisemigroups. Intermediate language terms are reified versions of these mathematical objects. In Section 4.5.3 we show how to give meaning to IRL terms by using a semantic evaluation function $[\![\_]\!]$ to translate from IRL terms into D.

Figure 4.1: Relationship between the metalanguage (RAML), the intermediate language (IRL) and the target language (C++).

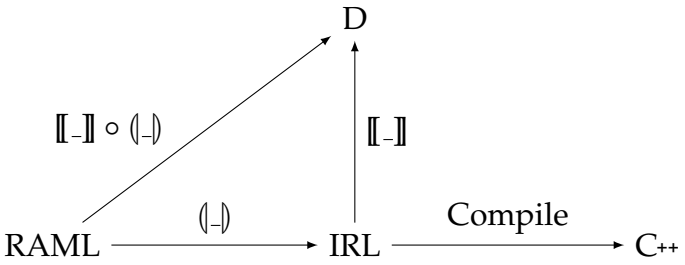


Figure 4.2: Use of the semantic domain D and semantic function $[\![\_]\!]$ to formally relate the metalanguage (RAML) and the intermediate language (IRL).

We can therefore establish the semantics of RAML terms by composing the translation function with the semantic evaluation function i.e. $[\![\_]\!] \circ (|\_|)$. Figure 4.2 illustrates these relations.

Our use of a semantic domain has several benefits. Firstly, and perhaps most importantly, it permits a concise specification of the meaning of metalanguage terms. In particular, the intermediate language can be given a semantics that is independent of any particular compiler implementation or target language. This reduces the scope for error during implementation of a compiler for the language. Secondly, in order to reason about the algebraic properties of metalanguage terms, it is necessary to have an algebraic semantics. The semantic domain allows the semantics to be given explicitly.

In this particular chapter we describe the intermediate language IRL_1 and domain D_1. Terms from the restricted metalanguage RAML_1 of Chapter 5 are translated into IRL_1. Then, in Chapter 7 we describe the extended intermediate language IRL_2 and domain D_2. These are a strict superset of the previous, and are used for assigning a meaning to the extended metalanguage RAML_2.

4.2 Basic definitions

4.2.1 Sets

In this section we define the notation that we use for representing sets. We denote the one element set {1} by 1, relying upon context to distinguish between the set and the element. Similarly, we denote the two element set $\{\top, \bot\}$ by 2. Informally, the symbol $\top$ corresponds to the value 'true' whilst the symbol $\bot$ corresponds to the value 'false'.

Description	Notation	Definition
Unit	1	$\{1\}$
Booleans	2	$\{\top, \bot\}$
Integers	$\mathbb{Z}$	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
Non-negative integers	$\mathbb{N}_0$	$\{0, 1, 2, \ldots\}$
Positive integers	$\mathbb{N}_1$	$\{1, 2, \ldots\}$
Integer range	$[n \,..\, m]$	$\{n, n+1, \ldots, m\}$
Strings	S^*	$\{s_1 \ldots s_n \mid s_i \in S\}$
Sets	$\mathcal{P}(S)$	$\{S' \mid S' \subseteq S\}$
Lists	$\mathcal{L}(S)$	$\{[s_1, \ldots, s_n] \mid n \in \mathbb{N} \land s_i \in S\}$
Simple lists	$\mathcal{S}(S)$	$\{[s_1, \ldots, s_n] \in \mathcal{L}(S) \mid s_i = s_j \Rightarrow i = j\}$

Figure 4.3: Notation for sets

The symbols $\mathbb{Z}$, $\mathbb{N}_0$ and $\mathbb{N}_1$ correspond to the usual subsets of the integers. We use the notation $[n \,..\, m]$ to denote integer ranges between n and m, with the requirement that $n \leq m$.

For a set S, denote the set of finite strings over S by S^*. Denote the set of finite subsets of S by $\mathcal{P}(S)$. Let $\mathcal{L}(S)$ denote the set of finite lists with elements from S. Finally, let a simple list be a list that contains no duplicate elements. Denote the set of finite, simple lists with elements from S by $\mathcal{S}(S)$. We summarise this notation in Figure 4.3. Note that for a list $X = [x_1, \ldots, x_n]$, we write $|X| = n$ to denote the length of the list, and $x_i \in X \iff 1 \leq i \leq n$ to denote the set membership predicate.

We now define the *disjoint union* construction, which is used to combine sets whilst maintaining the identity of the originating set for each element. Let $\mathcal{I} = \{i_1, \ldots, i_n\}$ be an *index set*. For sets $S_{i_1}, \ldots, S_{i_n}$, define the disjoint union $S_{i_1} + \cdots + S_{i_n}$ as

$$(\{i_1\} \times S_{i_1}) \cup \cdots \cup (\{i_n\} \times S_{i_n})$$

In the case where there are two non-indexed sets S and T, define the disjoint union $S + T$ as

$$(\{\text{inl}\} \times S) \cup (\{\text{inr}\} \times T).$$

The implicit indices 'inl' and 'inr' are mnemonics for 'in left' and 'in right', representing whether an element comes from the left set (S) or the right set (T) respectively.

4.2.2 Errors

Suppose that we have a binary operator $\oplus \in S \times S \to 1 + S$. For our purposes, we shall interpret the output value inl(1) as an error condition, such as a bounded list exceeding its maximum permitted length (here, $\oplus$ might be the list append operator). We now

modify this operator so that it can additionally handle the error as an input, directly propagating it to the output. Define the *lifted* operator $\hat{\oplus} \in (1 + S) \times (1 + S) \to 1 + S$ as

$$\begin{array}{rcll} \mathrm{inr}(x) & \hat{\oplus} & \mathrm{inr}(y) & = & x \oplus y \\ \mathrm{inl}(1) & \hat{\oplus} & \_ & = & \mathrm{inl}(1) \\ \_ & \hat{\oplus} & \mathrm{inl}(1) & = & \mathrm{inl}(1). \end{array}$$

The lifted operator has the infinity inl(1) as required, but otherwise behaves as the original operator for values in S. Providing that the lifted operator is associative, then the structure $(1 + S, \hat{\oplus})$ forms a semigroup. The lifting construction provides a convenient manner in which to separate the definition of the computational behaviour of the operator from its input error handling. As an aside, this construction is analogous to the bind operation in the 'Maybe' monad of the Haskell [96] functional programming language.

4.3 Semigroups

In this section we define the semigroups in D_1. We describe two kinds of objects: *base* algebras and *constructors*. Base algebras are atomic objects (in this case, semigroups) that cannot be decomposed, whilst constructors allow existing algebras to be combined to generate new algebras (again, in this case, semigroups).

4.3.1 Base semigroups

Conjunction and disjunction Define the *conjunctive* semigroup as $(2, \wedge)$ and the *disjunctive* semigroup as $(2, \vee)$, where the binary operators $\wedge$ and $\vee$ are the usual boolean conjunction and disjunction operators respectively.

Minimum and maximum Let $S \subseteq \mathbb{Z}$. Define the *minimum* and *maximum* semigroups over S as $(S, \min)$ and $(S, \max)$ respectively, where min and max are the usual minimum and maximum binary operators

Addition and multiplication Suppose that S is one of $\mathbb{N}_0$, $\mathbb{N}_1$ or $\mathbb{Z}$. Define the additive and multiplicative semigroups as $(\mathbb{N}, +)$ and $(\mathbb{N}, \times)$, where $+$ and $\times$ are the usual addition and multiplication operators. Now suppose that S is the set $[n \,..\, m]$. We face the problem that in general S is not *closed* under addition and multiplication. That is, for $s_1, s_2 \in [n \,..\, m]$, it is possible that $s_1 + s_2 \notin [n \,..\, m]$. We address this problem by instead generating the error 1 in those situations where the result is not a member of $[n \,..\, m]$. That is, we define the bounded addition operator $+ \in S \times S \to 1 + S$ as

$$s_1 + s_2 = \begin{cases} (\mathrm{inr}, s_1 + s_2) & n \leq (s_1 + s_2) \leq m \\ (\mathrm{inl}(1) & \text{otherwise,} \end{cases}$$

where the additions on the right of the definition are the usual 'platonic' versions. We lift this operator to a semigroup as $(1 + S, \hat{+})$. Define bounded multiplication similarly.

Set union and intersection Let S be a set. Define the union and intersection semigroups as $(\mathcal{P}(S), \cup)$ and $(\mathcal{P}(S), \cap)$, where $\cup$ and $\cap$ are the usual set union and intersection operators.

List append Let S be a set. Let $s_1, s_2 \in \mathcal{L}(S)$ with $s_1 = [s_{1,1}, \ldots, s_{m,1}]$ and $s_2 = [s_{1,2}, \ldots, s_{n,2}]$. Define the list append operator as

$$\begin{aligned} s_1 \mathbin{@} s_2 &= [s_{1,1}, \ldots, s_{m,1}] \mathbin{@} [s_{1,2}, \ldots, s_{n,2}] \\ &= [s_{1,1}, \ldots, s_{m,1}, s_{1,2}, \ldots, s_{n,2}]. \end{aligned}$$

Using this operator, define the list append semigroup over S as $(\mathcal{L}(S), @)$. We now turn to the append operation over simple lists $s_1, s_2 \in \mathcal{S}(T)$. We close the set under the append operation by generating an error if there is a duplicate element. That is, define the overloaded operator $@ \in \mathcal{S}(S) \times \mathcal{S}(S) \to 1 + \mathcal{S}(S)$ as

$$s_1 \mathbin{@} s_2 = \begin{cases} \mathrm{inr}(s_1 \mathbin{@} s_2) & s_1 \cap s_2 = \emptyset \\ \mathrm{inl}(1) & \text{otherwise,} \end{cases}$$

where the appends on the right of the definition are the usual platonic versions. Then define the list append semigroup over S as $(1 + \mathcal{S}(S), \hat{@})$.

Left and right Let S be a set. Define the *left* semigroup as (S, left), where for $s_1, s_2 \in S$, $\mathrm{left}(s_1, s_2) = s_1$. Define the *right* semigroup as (S, right), where $\mathrm{right}(s_1, s_2) = s_2$.

4.3.2 Semigroup constructors

Lexicographic list Suppose that T is $\mathcal{L}(S)$, with $S = (S, \oplus)$ a selective semigroup. For $t_1, t_2 \in T$, with $t_1 = [t_{1,1}, \ldots, t_{m,1}]$, $t_2 = [t_{1,2}, \ldots, t_{n,2}]$ and $k = \min(m, n)$, define the operator

$$t_1 \circledast t_2 = \begin{cases} t_1 & t_1 = t_2 \\ t_1 & \exists i \in \{1 \mathinner{..} k\}. \forall j < i. t_{j,1} = t_{j,2} \wedge t_{i,1} = t_{i,1} \oplus t_{i,2} \neq t_{i,2} \\ t_2 & \exists i \in \{1 \mathinner{..} k\}. \forall j < i. t_{j,1} = t_{j,2} \wedge t_{i,1} \neq t_{i,1} \oplus t_{i,2} = t_{i,2} \\ t_1 & \forall i \in \{1 \mathinner{..} k\}. t_{i,1} = t_{i,2} \wedge |t_1| < |t_2| \\ t_2 & \forall i \in \{1 \mathinner{..} k\}. t_{i,1} = t_{i,2} \wedge |t_2| < |t_1|. \end{cases}$$

This operator selects the list that is lexicographically minimal according to S. We form a semigroup as $(\circledast, T)$. Now suppose that T is $\mathcal{S}(S)$. In this case, we close T by adding the error value 1 in the usual manner.

Alpha and omega Let $S = (S, \oplus_S)$ be a semigroup. We show how to lift the semigroup S to operate over the set $1 + S$. We can either treat the value $\mathrm{inl}(1)$ as an identity or as

an infinity. For the former, define the semigroup $\alpha(S) = (1 + S, \oplus)$, where

$$\begin{array}{rcl} \mathrm{inr}(s_1) \oplus \mathrm{inr}(s_2) & = & \mathrm{inr}(s_1 \oplus_S s_2) \\ \mathrm{inl}(1) \oplus \mathrm{inr}(s_2) & = & \mathrm{inr}(s_2) \\ \mathrm{inr}(s_1) \oplus \mathrm{inl}(1) & = & \mathrm{inr}(s_1) \\ \mathrm{inl}(1) \oplus \mathrm{inl}(1) & = & \mathrm{inl}(1). \end{array}$$

For the latter, define the semigroup $\omega(S) = (1 + S, \oplus)$, where

$$\begin{array}{rcl} \mathrm{inr}(s_1) \oplus \mathrm{inr}(s_2) & = & \mathrm{inr}(s_1 \oplus_S s_2) \\ \mathrm{inl}(1) \oplus \_ & = & \mathrm{inl}(1) \\ \_ \oplus \mathrm{inl}(1) & = & \mathrm{inl}(1). \end{array}$$

These constructors are useful when defining bisemigroups in which one of the operators can produce an error. They allow the other operator to be lifted to accommodate the error, either treating it as an identity or as an infinity.

Direct product Let $S = (S, \oplus_S)$ and $T = (T, \oplus_T)$ be semigroups. Define the *direct product* of S and T as $S \times T = (S \times T, \oplus_{S \times T})$, where for $(s_1, t_1), (s_2, t_2) \in S \times T$,

$$(s_1, t_1) \oplus_{S \times T} (s_2, t_2) = (s_1 \oplus_S s_2, t_1 \oplus_T t_2)$$

This semigroup is used for modelling paired, yet independent computations. Note that we can extend this operation to the n-ary case in the obvious manner.

Lexicographic product Let $S = (S, \oplus_S)$ and $T = (T, \oplus_T)$ be semigroups, with both selective. Define the *lexicographic product* of S and T as $S \vec{\times} T = (S \times T, \oplus_{S \vec{\times} T})$, where for $(s_1, t_1), (s_2, t_2) \in S \times T$,

$$(s_1, t_1) \oplus_{S \vec{\times} T} (s_2, t_2) = \begin{cases} (s_1, t_1 \oplus_T t_2) & s_1 = s_2 \\ (s_1, t_1) & s_1 = s_1 \oplus_S s_2 \neq s_2 \\ (s_2, t_2) & s_2 \neq s_1 \oplus_S s_2 = s_2. \end{cases}$$

This semigroup is described in more detail in [17]. It is used for modelling ordered choice; when combining values, priority is given to the S component. The T operator is only used if there is a 'tie' in the S component. Again, we can extend this operation to the n-ary case in the obvious manner.

Disjoint union Suppose that we have a collection of semigroups $S_1, \ldots, S_n$ with $S_i = (S_i, \oplus_i)$. Let $S = S_1 + \cdots + S_n$. Define the operator $\oplus \in S \times S \to 1 + S$ as

$$(l_i, x) \oplus (l_j, y) = \begin{cases} \mathrm{inr}(l_i, x \oplus_i y) & i = j \\ \mathrm{inl}(1) & \text{otherwise.} \end{cases}$$

Finally, lift this operator to obtain the *disjoint union* semigroup $S_1 + \cdots + S_n = (1 + S, \hat{\oplus})$. The disjoint union semigroup is used to model situations where there may be several different types of values (such as metrics that are either internal or external to an AS). If we attempt to combine values of different types, then it is necessary to generate an error, as above.

4.3.3 Properties

Recall that we define a selection of algebraic properties for semigroups in Figure 2.10(a). It is straightforward to calculate such properties for the base semigroups. We list a number of base semigroups and their associated properties at the end of this chapter, in Figure 4.9. We also require rules to specify when properties hold of constructed semigroups. For example, suppose that we have semigroups $S = (S, \oplus)$ and $T = (T, \oplus)$. The rule for commutativity of the semigroup $S \times T$ is given as

$$\textsc{comm}(S \times T) \iff \textsc{comm}(S) \wedge \textsc{comm}(T).$$

Therefore, in order for the direct product semigroup $S \times T$ to be commutative, it is necessary and sufficient for both S and T to be commutative.

Note that it is desirable to have 'if and only if' rules. That is, if a property does *not* hold of a constructed semigroup, then it should be possible to tell why this is the case. The direct product commutativity rule is an example of such a rule; if it is not commutative, then we know that at least one of the argument semigroups lacks the necessary commutativity property. In general, such information is useful for finding alternative semigroups in which the required properties *do* hold.

Currently, we do not know 'if and only if' rules for all constructions. Discovering and verifying such property rules is an area of active research. Early work suggests that some of this process may be amenable to automated reasoning.

4.4 Bisemigroups

In this section we define the bisemigroups in D_1. Analogously to semigroups, we define the base bisemigroups and then the bisemigroup constructors.

4.4.1 Base bisemigroups

And-or Define the semigroup comprising the boolean conjunction and disjunction operators as $(2, \wedge, \vee)$.

Max-min Let S be one of $\mathbb{Z}$, $\mathbb{N}_0$, $\mathbb{N}_1$ or $[n\,..\,m]$. Define the bisemigroup comprising the maximum and minimum operators over S as $(S, \max, \min)$.

Min-plus Let S be one of $\mathbb{Z}$, $\mathbb{N}_0$, $\mathbb{N}_1$. Define the semigroup comprising the minimum and addition operators over S as $(\mathbb{N}, \min, +)$. Now suppose that S is $[n\,..\,m]$. In this situation it is possible to obtain an error 1 from the addition operator, and therefore we must lift the min operator to treat this value as an identity. That is, define the bisemigroup as $(1 + S, \alpha(\min), \hat{+})$.

Min-times Let S be one of $\mathbb{Z}$, $\mathbb{N}_0$, $\mathbb{N}_1$. Define the semigroup comprising the minimum and multiplication operators over S as $(\mathbb{N}, \min, \times)$. Now suppose that S is $[n \mathinner{..} m]$. In common with the min-plus bisemigroup, we must lift the min operator to treat the error value 1 as an identity. That is, define the bisemigroup as $(1 + S, \alpha(\min), \hat{\times})$.

Union-intersection Let S be a set. Define the bisemigroup comprising the union and intersection operators over $\mathcal{P}(S)$ as $(\mathcal{P}(S), \cup, \cap)$.

4.4.2 Bisemigroup constructors

Lexicographic list Let T be the set $\mathcal{L}(S)$, with $S = (S, \oplus)$ a selective semigroup. Define the *lexicographic list* bisemigroup as $(S, \circledast, @)$. Now suppose that T is $\mathcal{S}(S)$. In this case, the append operator must be lifted to treat the error value 1 as an infinity i.e. we have obtained the bisemigroup $(1 + T, \circledast, \omega(@))$.

Alpha/omega Let $S = (S, \oplus, \otimes)$ be a bisemigroup. Define the bisemigroup that lifts S to add 1 as the identity for $\oplus$ and the infinity for $\otimes$ as $(1 + S, \alpha(\oplus), \omega(\otimes))$.

Omega/alpha Let $S = (S, \oplus, \otimes)$ be a bisemigroup. Similarly to the previous constructions, define the bisemigroup that lifts S to add 1 as the infinity for $\oplus$ and the identity for $\otimes$ as $(1 + S, \omega(\oplus), \alpha(\otimes))$.

Direct product Let $S = (S, \oplus_S, \otimes_S)$ and $T = (T, \oplus_T, \otimes_T)$ be bisemigroups. Define the *direct product* of S and T as

$$S \times T = (S \times T, \oplus_{S\times T}, \otimes_{S\times T}),$$

Note that we can extend this operation to the n-ary case in the obvious manner.

Lexicographic product Let $S = (S, \oplus_S, \otimes_S)$ and $T = (T, \oplus_T, \otimes_T)$ be bisemigroups, with the additive components both selective. Define the *lexicographic product* of S and T as

$$S \vec{\times} T = (S \times T, \oplus_{S\vec{\times}T}, \otimes_{S\times T}),$$

Again, note that we can extend this operation to the n-ary case in the obvious manner.

Disjoint union Suppose that we have a collection of bisemigroups $S_{l_1}, \ldots, S_{l_n}$ with $S_{l_i} = (S_{l_i}, \oplus_{l_i}, \otimes_{l_i})$. Let $S = S_{l_1} + \cdots + S_{l_n}$. Define the operator $\oplus \in S \times S \to 1 + S$ as

$$(l_i, x) \oplus (l_j, y) = \begin{cases} \mathrm{inr}(l_i, x \oplus_i y) & i = j \\ \mathrm{inl}(1) & \text{otherwise.} \end{cases}$$

Define the operator $\otimes \in S \times S \to 1 + S$ similarly, mutatis mutandis. Finally, lift this operator to obtain the disjoint union bisemigroup $S_1 + \cdots + S_n = (1 + S, \hat{\oplus}, \hat{\otimes})$.

4.4.3 Properties

Recall that we define a number of algebraic properties of bisemigroups in Figure 2.10(b). It is relatively straightforward to calculate such properties for the base bisemigroups. In common with semigroups, we list a number of bisemigroups and their associated properties at the end of this chapter, in Figure 4.10.

The rules for calculating properties of constructed bisemigroups are more 'interesting' than those for semigroups. This is due to the presence of properties governing the interaction of the additive and multiplicative components of bisemigroups. As an example, we briefly describe the rule for distributivity of the bisemigroup lexicographic product. This definition is from [17]. Suppose that we have two bisemigroups, $S = (S, \oplus_S, \otimes_S)$ and $T = (T, \oplus_T, \otimes_T)$. The rule for left-distribution of the bisemigroup $S \vec{\times} T$ is given as:

$$\begin{aligned} &\textsc{left\_dist}(S \vec{\times} T) \iff \\ &\quad \textsc{left\_dist}(S) \wedge \textsc{left\_dist}(T) \wedge (\textsc{left\_canc}(S, \otimes_S) \vee \textsc{left\_const}(T, \otimes_T)) \end{aligned}$$

That is, we require both S and T to be distributive, and either the multiplicative component of S to be cancelative or else the multiplicative component of T to be constant. Note that this is an 'if and only if' rule, and therefore if $S \vec{\times} T$ is not distributive, then we can precisely determine why this is the case.

We now illustrate the application of this rule. Consider the pair of bisemigroups $S = (\mathbb{N}_0, \min, +)$ and $T = (\mathbb{N}_0, \max, \min)$. We have that $S \vec{\times} T$ is left-distributive because both S and T are left-distributive, and S is left-cancelative. However, $T \vec{\times} S$ is *not* left-distributive, because it is not that case that either T is left-cancelative or S is left-constant. The properties of these semigroups and bisemigroups are given at the end of this chapter, in Figures 4.9 and 4.10.

4.5 Intermediate language

In this section we define the intermediate language, IRL_1. This is a language based upon the semigroup and bisemigroup constructions from Sections 4.3 and 4.4. We translate routing language specifications written in our metalanguage RAML_1 (Chapter 5) into terms in IRL_1. As previously discussed, the intermediate language both gives a clear semantics to the metalanguage and also acts as a bridge between the high-level metalanguage constructions and the lower-level C++ output of the compiler.

We commence, in Section 4.5.1, by giving the lexical conventions that we use to define IRL_1. We use the same conventions when defining the metalanguage. Next, in Section 4.5.2, we define the syntax of IRL_1. Finally, in Section 4.5.3, we assign a semantics to IRL_1. This maps syntax from IRL_1 into the semantic domain D_1.

$$
\begin{array}{rcll}
l & ::= & \texttt{A} \ldots \texttt{Z} \mid \texttt{a} \ldots \texttt{z} & \text{(letter)} \\
d & ::= & \texttt{0} \ldots \texttt{9} & \text{(digit)} \\
c & ::= & l \mid d \mid \texttt{␣} \mid \texttt{!} \mid \texttt{\#} \mid \texttt{\$} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{'} \mid \texttt{(} \mid \texttt{)} \mid \texttt{*} \mid \texttt{+} \mid \texttt{,} \mid \texttt{-} \mid \texttt{.} \mid \texttt{/} & \text{(character)} \\
 & & \mid \texttt{:} \mid \texttt{;} \mid \texttt{<} \mid \texttt{=} \mid \texttt{>} \mid \texttt{?} \mid \texttt{@} \mid \texttt{[} \mid \texttt{\textbackslash} \mid \texttt{]} \mid \texttt{\^{}} \mid \texttt{\_} \mid \texttt{`} \mid \texttt{\{} \mid \texttt{|} \mid \texttt{\}} \mid \texttt{\~{}} & \\
n, m & ::= & [\texttt{-}]\, d\, \{ d \} & \text{(integer)} \\
s & ::= & \texttt{"}\{c\}\texttt{"} & \text{(string)} \\
i & ::= & (\texttt{\_} \mid l)\, \{ \texttt{-} \mid \texttt{\_} \mid l \mid d \} & \text{(identifier)}
\end{array}
$$

Figure 4.4: Syntax of basic lexical classes

4.5.1 Lexical conventions

In this section we briefly explain the lexical conventions that we use to define the syntax of IRL_1. Note that these conventions are based upon those from [97]. Terminal symbols are written in typewriter font. For example, the unit type is written `unit`. Non-terminal symbols are written in italicised font. For example, the syntactic category of value is written v. Braces $\{\ldots\}$ denote zero or more repetitions of the enclosed symbols, whist the variant $\{\ldots\}^+$ denotes one or more repetitions. Square brackets $[\ldots]$ denote optional symbols. Parentheses $(\ldots)$ are used for grouping. We define some basic syntactic classes in Figure 4.4. These grammars are written in Backus-Naur Form (BNF). Note that characters c are drawn from the printable subset of ASCII, with double quotation marks removed.

4.5.2 Syntax

The intermediate language contains types, values, semigroups and bisemigroups. We list the syntax of types *ty* and values v in Figure 4.5. The language contains *scalar* values, such as integers and strings, and *vector* values such as lists and sets. Formally, as we show in the next section, types correspond to sets whilst values are elements of these sets.

We now turn to the computational content of the language: seimgroups and bisemigroups. We list the syntax of semigroups and bisemigroups in Figure 4.6. Syntactically, semigroups *sg* comprise pairs `(`*ty*`,` *bo*`)`, where *bo* is a binary operator over that type. For example, `(int, plus)` is a semigroup. Bisemigroups *bsg* are similar to semigroups, but contain an additional binary operator. They are written as triples `(`*ty*`,` *bo*`,` *bo*`)`. An example of a bisemigroup is `(int, min, plus)`.

4.5.3 Semantics

It is relatively straightforward to assign a semantics to IRL_1. This is because the language is a reified version of the algebraic constructions given in the earlier parts of

ty	::=		(type)
	|	`bool`	(boolean)
	|	ty_num_+	(extended numeric)
	|	`string`	(string)
	|	`set(`ty`)`	(set)
	|	`list(`ty`)`	(list)
	|	`list_simp(`ty`)`	(simple list)
	|	`rec((`i_1`,`ty_1`),...,(`i_n`,`ty_n`))`	(product)
	|	`disj_union((`i_1`,`ty_1`),...,(`i_n`,`ty_n`))`	(disjoint union)
	|	`add_const(`i`,`ty`)`	(add constant)
ty_num_+	::=		(extended numeric)
	|	ty_num	(numeric)
	|	`int_bound(`n`,`m`)`	(bounded integer)
ty_num	::=		(numeric)
	|	`int`	(integer)
	|	`int_non_neg`	(non-negative integer)
	|	`int_pos`	(positive integer)
v	::=		(value)
	|	`unit`	(unit)
	|	`true | false`	(booleans)
	|	`int(`n`)`	(integer)
	|	`string(`s`)`	(string)
	|	`{`v_1`,...,`v_n`}`	(set)
	|	`[`v_1`,...,`v_n`]`	(list)
	|	`(`v_1`,...,`v_n`)`	(product)
	|	`inj(`i`,`v`)`	(disjoint union)

Figure 4.5: Syntax of IRL_1 types and values

this chapter. We assign a semantics to IRL_1using a *denotational* approach [98], whereby each syntactic category is embedded into a semantic domain. In our case, the semantic domain is D_1, and for a given syntactic construct x in IRL_1, we write $[\![x]\!]$ to denote the corresponding object in D_1. This approach contrasts with the *operational* approach [99], whereby the meaning of programs is specified using rules over the syntax.

We give the denotations of types and values in Figures 4.7. Types correspond to sets, whilst values are members of sets. We say that a value v is of type ty if and only if $[\![v]\!] \in [\![ty]\!]$. We also say that a type ty_1 is a *sub-type* of ty_2 if and only if $[\![ty_1]\!] \subseteq [\![ty_2]\!]$. For example, `int_pos` is a sub-type of `int`. Due to the presence of sub-types, a value can be of multiple types. For example, the integer `14` is of type `int_pos` and `int`. This does not cause any difficulties, because we are only interested in the *type checking* problem whereby we verify whether a value is of a particular type. This contrasts with the *type*

bo	::=		(binary operator)
	|	`and`	(boolean conjunction)
	|	`or`	(boolean disjunction)
	|	`min`	(integer minimisation)
	|	`max`	(integer maximisation)
	|	`plus`	(integer addition)
	|	`times`	(integer multiplication)
	|	`union`	(set union)
	|	`inter`	(set intersection)
	|	`app`	(list append)
	|	`list_lex(`*bo*`)`	(lexicographic list)
	|	`left`	(left)
	|	`right`	(right)
	|	`alpha(`*bo*`)`	(alpha)
	|	`omega(`*bo*`)`	(omega)
	|	`dir_prod(`bo_1`,...,`bo_n`)`	(direct product)
	|	`lex_prod(`bo_1`,,...`bo_n`)`	(lexicographic product)
	|	`disj_union(`bo_1`,,...`bo_n`)`	(disjoint union)
sg	::=	`(`*ty*`, `*bo*`)`	(semigroup)
bsg	::=	`(`*ty*`, `*bo*`, `*bo*`)`	(bisemigroup)

Figure 4.6: Syntax of IRL_1 semigroups and bisemigroups

inference problem, where we must synthesise the type of a value.

The denotations of (syntactic) semigroups are given in Figure 4.8. The denotation of a syntactic semigroup `(`*ty*`, `*bo*`)` is a semigroup $(S, \oplus)$. The denotation of syntactic bisemigroup `(`*ty*`, `bo_1`, `bo_2`)` is obtained by taking the denotations $(S_1, \oplus) = [\![\texttt{(}ty\texttt{, } bo_1\texttt{)}]\!]$ and $(S_2, \otimes) = [\![\texttt{(}ty\texttt{, } bo_2\texttt{)}]\!]$, and then combining to form the bisemigroup $(S, \oplus, \otimes)$ with $S = S_1 = S_2$.

We note that the meaning of a binary operator *bo* is dependent upon the type *ty* that it operates upon. This is known as *overloading*. For example, the semigroups `(int, plus)` and `(disj_union(unit, int_bound(1,16)), plus)` both contain the identical syntactic operator `plus`, yet this operator has different denotations for each semigroup. We chose to overload operators, instead of having multiple different versions, to minimise the size of the intermediate language.

Type ty	Denotation $[\![ty]\!]$
`unit`	1
`bool`	2
`int`	$\mathbb{N}$
`int_non_neg`	$\mathbb{N}_0$
`int_pos`	$\mathbb{N}_1$
`int_bound(`n`,`m`)`	$[n \mathinner{..} m]$
`string`	c^*
`set(`ty`)`	$\mathcal{P}([\![ty]\!])$
`list(`ty`)`	$\mathcal{L}([\![ty]\!])$
`list_simp(`ty`)`	$\mathcal{S}([\![ty]\!])$
`rec((`i_1`,`ty_1`),...,(`i_n`,`ty_n`))`	$[\![ty_1]\!] \times \cdots \times [\![ty_n]\!]$
`disj_union((`i_1`,`ty_1`),...,(`i_n`,`ty_n`))`	$[\![ty_1]\!] + \cdots + [\![ty_n]\!]$
`add_const(`i`,`ty`)`	$1 + [\![ty]\!]$

Value v	Denotation $[\![v]\!]$
`unit`	1
`true`	$\top$
`false`	$\bot$
`int(`n`)`	n
`string(`s`)`	s
`{`v_1`,...,`v_n`}`	$\{[\![v_1]\!], \ldots, [\![v_n]\!]\}$
`[`v_1`,...,`v_n`]`	$[[\![v_1]\!], \ldots, [\![v_n]\!]]$
`(`v_1`,...,`v_n`)`	$([\![v_1]\!], \ldots, [\![v_n]\!])$
`inj(`i`,`v`)`	$(i, [\![v]\!])$

Figure 4.7: Denotations of IRL_1 types and values

Semigroup sg	Denotation $[\![sg]\!]$
`(bool, and)`	$(2, \wedge)$
`(bool, or)`	$(2, \vee)$
`(`ty_num_+`, min)`	$([\![ty\_num_+]\!], \min)$
`(`ty_num_+`, max)`	$([\![ty\_num_+]\!], \max)$
`(`ty_num`, plus)`	$([\![ty\_num]\!], +)$
`(disj_union(unit, int_bound(`n`,`m`)), plus)`	$(1 + [n\,..\,m], \hat{+})$
`(`ty_num`, times)`	$([\![ty\_num]\!], \times)$
`(disj_union(unit, int_bound(`n`,`m`)), times)`	$(1 + [n\,..\,m], \hat{\times})$
`(set(`ty`), union)`	$(\mathcal{P}([\![ty]\!]), \cup)$
`(set(`ty`), inter)`	$(\mathcal{P}([\![ty]\!]), \cap)$
`(list(`ty`), app)`	$(\mathcal{L}([\![ty]\!]), @)$
`(disj_union(unit, list_simp(`ty`)), app)`	$(1 + \mathcal{S}([\![ty]\!]), \hat{@})$
`(list(`ty`), list_lex)`	$(\mathcal{L}([\![ty]\!]), \circledast)$
`(`ty`, left)`	$([\![ty]\!], \text{left})$
`(`ty`, right)`	$([\![ty]\!], \text{right})$
`(disj_union(unit, `ty`), alpha(`bo`))`	$(1 + S, \alpha(\oplus))$ where $(S, \oplus) = [\![(ty,\ bo)]\!]$
`(disj_union(unit, `ty`), omega(`bo`))`	$(1 + S, \omega(\oplus))$ where $(S, \oplus) = [\![(ty,\ bo)]\!]$
`(prod(`$ty_1,\ldots,ty_n$`), dir_prod(`$ty_1,\ldots,ty_n$`))`	$S_1 \times \cdots \times S_n$ where $(S_i, \oplus_i) = [\![(ty_i,\ bo_i)]\!]$
`(prod(`$ty_1,\ldots,ty_n$`), lex_prod(`$ty_1,\ldots,ty_n$`))`	$S_1 \vec{\times} \cdots \vec{\times} S_n$ where $(S_i, \oplus_i) = [\![(ty_i,\ bo_i)]\!]$
`(disj_union(unit, union(`$ty_1,\ldots,ty_n$`)), disj_union(`$bo_1,\ldots,bo_n$`))`	$S_1 + \cdots + S_n$ where $(S_i, \oplus_i) = [\![(ty_i,\ bo_i)]\!]$

Figure 4.8: Denotations of IRL_1semigroups

Semigroup	COMM	SEL	IDEM	LEFT_CANC	RIGHT_CANC	LEFT_CONST	RIGHT_CONST	ALPHA	OMEGA
$(\mathbb{B}, \wedge)$	✓	✓	✓	×	×	×	×	true	false
$(\mathbb{B}, \vee)$	✓	✓	✓	×	×	×	×	false	true
$(\mathbb{N}_0, \min)$	✓	✓	✓	×	×	×	×	×	0
$(\mathbb{N}_0, \max)$	✓	✓	✓	×	×	×	×	0	×
$([n\,..\,m], \min)$	✓	✓	✓	×	×	×	×	m	n
$([n\,..\,m], \max)$	✓	✓	✓	×	×	×	×	n	m
$(\mathbb{Z}, +)$	✓	×	×	✓	✓	×	×	0	×
$(\mathbb{Z}, \times)$	✓	×	×	✓	✓	×	×	×	0
$(\mathcal{P}(S), \cup)$	✓	×	✓	×	×	×	×	$\emptyset$	S
$(\mathcal{P}(S), \cap)$	✓	×	✓	×	×	×	×	S	$\emptyset$
$(\mathcal{L}(S), @)$	×	×	×	✓	✓	×	×	[]	×
(S, left)	×	✓	✓	×	✓	✓	×	×	×
(S, right)	×	✓	✓	✓	×	×	✓	×	×

Figure 4.9: Algebraic properties of base D_1 semigroups

Bisemigroup	LEFT_DIST	RIGHT_DIST	LEFT_INCR	RIGHT_INCR	ALPHA_OMEGA	OMEGA_ALPHA
$(\mathbb{B}, \wedge, \vee)$	✓	✓	✓	✓	true	false
$(\mathbb{N}_0, \max, \min)$	✓	✓	×	×	0	×
$(\mathbb{N}_0, \min, +)$	✓	✓	✓	✓	×	0
$(1 + [n\,..\,m], \alpha_c(\min), \hat{+}),\ n > 0$	✓	✓	✓	✓	inl(1)	×
$(1 + [n\,..\,m], \alpha_c(\min), \hat{+}),\ n = 0$	✓	✓	✓	✓	inl(1)	inr(0)
$(\mathcal{P}(S), \cup, \cap)$	✓	✓	✓	✓	$\emptyset$	S

Figure 4.10: Algebraic properties of base D_1 bisemigroups

Chapter 5

RAML_1: Mini metalanguage

In this chapter we describe RAML_1, the *mini metalanguage*. This is a restricted version of the *extended metalanguage*, RAML_2, that we will describe in Chapter 7. RAML_1 can be used to define routing algebras based upon semigroups and bisemigroups, whereas RAML_2 extends RAML_1 to additionally include preorders and transforms. We have separated the language into restricted and extended versions purely to simplify the presentation.

We commence this chapter with an example to give an intuition for the metalanguage (§ 5.1). We then formally describe the metalanguage (§ 5.2). Finally, we give a translation into the intermediate language (§ 5.3). This both indirectly defines the semantics of the metalanguage and also describes the basis from which C++ code is generated.

5.1 Example

Prior to describing RAML_1, we give an example routing language specification written in the metalanguage. Our intention is to give a guiding intuition for the language without overwhelming the reader with the technical details.

Suppose that we wish to compute 'shortest-paths'. That is, links are weighted with non-negative integer values, and we add these weights together to obtain path weights, with smaller weights preferred. Recall from Section 2.6 that we can algebraically model this structure using the min-plus semiring. In RAML_1, we write this structure as `min_plus(int_non_neg)`. The type argument `int_non_neg` establishes that the minimisation and addition is occurring over the non-negative subset of the integers.

Now suppose that we wish to tie-break upon bandwidth. That is, given a pair of shortest paths, we wish to use a bandwidth metric to decide between them. We compute bandwidth by again labelling arcs with non-negative integer values. However, the weight of a path is now the 'bottle-neck' (minimum) arc weight along that path, and

we prefer larger values. We algebraically model this structure using the max-min semiring. We write this structure in RAML$_1$ as `max_min(int_non_neg)`.

Finally, we wish to avoid transient looping behaviour when we use this routing language with a vectoring algorithm such as distributed Bellman-Ford. For this, we use a construction similar to the BGP AS path attribute whereby integer router identifiers are added to a path structure. We prevent looping by imposing the constraint that the path must contain no duplicate node identifiers. We write this in RAML$_1$ as `list_lex_app_simp(NOTSIMP, min(int_pos))`.

We combine the three structures using the lexicographic product construction. We write this in RAML$_1$ as

```
lex_prod(dist : min_plus(int_non_neg),
         bw   : max_min(int_non_neg),
         path : list_lex_app_simp(NOTSIMP, min(int_pos)))
```

The identifiers `dist`, `bw` and `path` are field labels. Labels are syntactic sugar to convey the meaning of particular values within a field, hence clarifying configuration.

Metrics and policy comprise triples <`dist=`n`, bw=`m`, path=[...]`>, where n and m are non-negative integers, and the path contains a sequence of comma-separated integers. We now illustrate policy application and metric summarisation. The metric <`dist=2, bw=10, path=[3,4]`> is more preferred than <`dist=5, bw=20, path=[5]`> because the former has a lower distance component. Setting the two distant components to be identical, we have that the metric <`dist=5, bw=10, path=[3,4]`> is *less* preferred than <`dist=5, bw=20, path=[5]`>. This is because the latter metric has a larger bandwidth component.

We now turn to the application of policy to metrics. Consider the application of the policy <`dist=3, bw=5, path=[1]`> to the metric <`dist=2, bw=10, path=[3,4]`>. This results in the metric <`dist=5, bw=5, path=[1,3,4]`>. That is, distances are added, bandwidths are minimised and paths are appended.

Using rules similar to those presented in Chapter 2, it is straightforward to show that the corresponding bisemigroup is both increasing and distributive. Therefore this routing language can be combined with either a vectoring or a link-state/Dijkstra algorithm. Furthermore, due to distributivity, routing solutions from a vectoring algorithm will be globally optimal. Now consider exchanging the order of the bandwidth and distance components to obtain the following routing language specification:

```
lex_prod(bw   : max_min(int_non_neg),
         dist : min_plus(int_non_neg),
         path : list_lex_app_simp(NOTSIMP, min(int_pos)))
```

We can automatically infer that the corresponding bisemigroup is increasing, but *not* distributive. Therefore the routing language can only be combined with a vectoring algorithm, but not a link-state Dijkstra algorithm. Furthermore, solutions will only be locally optimal in general.

5.2 Metalanguage

In this section we define RAML$_1$. We first give a brief overview of the language, before giving its full definition.

The language is declarative, meaning that terms with the language abstractly specify intended behaviours without defining the particular implementation technique. Tasks such as the selection of appropriate data-structures and algorithms are delegated to the compiler. This separation from the implementation details increases the clarity of the language and indeed allows the compiler a large degree of choice for efficient implementation. RAML$_1$ is also statically typed. This means that it is possible to syntactically eliminate a large class of errors at compile-time that would otherwise manifest themselves at runtime [99].

We define RAML$_1$ in two steps. Firstly, we define its syntax. Whilst this provides sufficient information for writing a routing language specification, it does not tell us what such a specification means. Therefore, the second component of the RAML$_1$ definition is a translation into the intermediate language IRL$_1$. As discussed, IRL$_1$ has a formal semantics, and in this way we assign meaning to the metalanguage.

We commence by defining the syntax of types and their associated values (§ 5.2.1). Note that we reuse the lexical conventions from Section 4.5.1. Next, we define the computational component of the language: semigroups and bisemigroups (§ 5.2.2). Finally, we give a translation from RAML$_1$ into IRL$_1$ (§ 5.3).

5.2.1 Types and values

In this section we define the object-level types and values of RAML$_1$. Types specify the permitted values that metrics and policy may take. Types are both explicitly written by users and are also automatically inferred at compilation. Semantically, types comprise *sets* of values. We note that the types described in this section occur *within* routing language specifications and are distinct from the types *of* specifications. Additionally, values do not occur within specifications, but are instead manipulated by the code generated from specifications. Therefore a rigorous definition of values is still essential for defining the language.

ty	::=		(type)
	|	`unit`	(unit)
	|	`bool`	(booleans)
	|	`int`	(integers)
	|	`int_non_neg`	(non-negative integers)
	|	`int_pos`	(positive integers)
	|	`int_bound(`n`,`m`)`	(bounded integers)
	|	`string`	(strings)
	|	`set(`ty`)`	(sets)
	|	`list(`ty`)`	(lists)
	|	`list_simp(`ty`)`	(simple lists)
	|	`rec(`i_1`:`ty_1`,...,`i_n`:`ty_n`)`	(records)
	|	`disj_union(`i_1`:`ty_1`,...,`i_n`:`ty_n`)`	(disjoint unions)
	|	`add_const(`i`,`ty`)`	(constants)
v	::=		(value)
	|	`unit`	(unit)
	|	`true | false`	(boolean)
	|	n	(integer)
	|	s	(string)
	|	`[`v_1`,...,`v_n`]`	(list)
	|	`{`v_1`,...,`v_n`}`	(set)
	|	$<i_1 = v_1, \ldots, i_n = v_n>$	(record)
	|	`inj(`i`,` v`)`	(disjoint union)
	|	i	(constant)

Figure 5.1: Syntax of RAML$_1$ types and values

RAML$_1$ contains scalar values such as integers and strings. The former is useful for representing values such as distance and bandwidth, whilst the latter is used for assigning names to objects such as routers etc. The language also contains vector values such as lists, sets and records (labelled n-ary tuples). Such values are essential for representing more complicated data-structures. For example, lists may be used to represent structures similar to AS paths.

In common with the intermediate language, a value may be a member of more than one type. For example, the type `int` corresponds the set of integers $\mathbb{Z}$ whilst `int_bound(`n`,`m`)` denotes the finite subset $[n \mathinner{..} m]$. Formally, `int_bound(`n`,`m`)` is a sub-type of `int`. Although the inclusion of sub-types increases the complexity of the language, this facility allows more constraints to be encoded within protocol specifications. In some cases, it is only possible to derive the required correctness properties if a restricted version of a type is used. For example, it is necessary to exclude the negative integers from the min-plus algebra in order to obtain an increasing algebra.

We give the syntax of RAML$_1$ types and values in Figure 5.1. We now describe the various types and values in more detail.

Unit The unit type `unit` contains a single constant, also written as `unit`. Note that this type carries no information. Instead, it is often used within a disjoint union to represent an error case.

Booleans The boolean type `bool` contains two constants, written as `true` and `false`.

Integers There are four *integral* types, each corresponding to subsets of the integers. The `int` type corresponds to the set of all integers, whilst the `int_non_neg` and `int_pos` types correspond to the non-negative and positive subsets of the integers respectively. Finally, the `int_bound(`n`,`m`)` type denotes a finite subset of integers in the range $[n, m]$, with $n \leq m$. Note the use of the brackets for parameters. Integer values are written in their base-10 representation, such as `16` or `-4`.

Strings The string type `string` comprises all finite sequences of characters. String values are written surrounded by double quotations, such as `"foo"`.

Lists The list type `list(`ty`)` denotes finite but unbounded lists of elements from ty. List values are written as sequences of comma-separated values, surrounded by square brackets. For example, the two-element list, of which the first element is the integer 16 and the second element is the integer -4, is written as `[16, -4]`.

Sets The set type `set(`ty`)` denotes finite but unbounded sets of elements from ty. Set values are written as sequences of comma-separated values, surrounded by braces. For example, the two-element set comprising the integers 16 and -4 could be written as `{16, -4}`, `{-4, 16}` or even `{16, -4, -4}`.

Records Records are essentially labelled products. The label names may be used to convey additional semantic information within specifications. For types $ty_1, \ldots, ty_n$ and distinct labels $i_1, \ldots, i_n$, we write the type `rec(`i_1`:`ty_1`,...,`i_n`:`ty_n`)` for the record with field i_1 of type ty_1 etc. Syntactically, record values are written as sequences of assignments of values to labels, surrounded by angle brackets. For example, the record with label `name` assigned the string `"foo"` and `dist` assigned the integer `45` is written as `<name="foo", dist=16>`. This has the type `rec(name:string, dist:int)`.

Disjoint unions The disjoint union type is used to represent collections of values of different types. For types $ty_1, \ldots, ty_n$ and distinct labels $i_1, \ldots, i_n$, we write the type `disj_union(`i_1`:`ty_1`,...,`i_n`:`ty_n`)` to denote n-way disjoint union, with tag i_1 corresponding to a value of type ty_1 etc. Disjoint union values correspond to pairs of a label and a value. They are written as an application of the `inj` (inject) function to a pair of a label and a value. For example, the integer 16 labelled with `dist` is written as `inj(dist, 16)`.

Constants For a type ty, and a constant i, the type `add_const(`i`, `ty`)` adds the constant i to ty. This construction is syntactic sugar for the type `disj_union(inl:`i`, inr:`ty`)`, with

the additional feature that value tags are omitted. This requires that no duplicate constants are added. The value `inj(inl, `i`)` is written as i and `inj(inr, `v`)` is written as v. For example, the values `omega` and `16` are both of the type `add_const(omega, int)`.

5.2.2 Semigroups and bisemigroups

In this section we define the 'syntactic' semigroups and bisemigroups of RAML$_1$. Semigroups and bisemigroups specify the computational behaviour of the language, defining how values are manipulated within routing protocols. For us, the values and types are those from Section 5.2.1. Whereas the syntactic semigroups and bisemigroups have implicit computational content, the 'semantic' versions in the intermediate language have explicit types and operators that are amenable to code generation.

The semigroup and bisemigroup operators are similar to those from the intermediate language. We specify the syntax of semigroups and bisemigroups in Figure 5.2. The correspondence with IRL$_1$ terms is fairly straightforward to infer, although it is formally specified in Section 5.3.

We briefly discuss several representative examples of RAML$_1$ semigroups. The `and` semigroup corresponds to the usual boolean conjunction. The `plus(`*ty_num*`)` semigroup corresponds to integer addition over the numeric type *ty_num*. The semigroup `plus_bound(`i`,`n`,`m`)` is similar, but instead defines bounded arithmetic over the integer range `int_bound(`n`,`m`)`, generating the error constant i if the resultant value is out of this range. The `app_simp(`i`,`*ty*`)` semigroup corresponds to the list append operation lifted to operate over the type `add_const(`i`, list_simp(`*ty*`))`. The error constant i is generated if there is a duplicate value in a list. The `add_alpha(`i`, `*sg*`)` semigroup lifts *sg* to operate over the type `add_const(`i`, `*ty*`)`, where *ty* is the type of *sg*. The resultant semigroup treats the constant i as an identity value. Finally, the `lex_prod(`i_1`:`sg_1`,...,`i_n`:`sg_n`)` semigroup corresponds to the n-ary semigroup lexicographic product.

Moving onto representative examples of RAML$_1$ bisemigroups, the `min_plus(`*ty_num*`)` bisemigroup corresponds to integer minimisation and addition over the numeric type *ty_num*, as illustrated in our introductory example (§ 5.1). The `min_plus_bound(`i`,`n`,`m`)` bisemigroup is similar, but instead defines bounded arithmetic over the integer range `int_bound(`n`,`m`)` where i is an error constant. The `twin(`*sg*`)` bisemigroup defines the bisemigroup where *sg* is used for both the additive and multiplicative semigroups. The `swap(`*bsg*`)` bisemigroup simply replaces the additive semigroup of *bsg* with the multiplicative semigroup, and vice versa. The `list_lex_app_simp(`i`, `*sg*`)` bisemigroup lexicographically summarises simple lists using the *sg* semigroup, and combines lists using the append operation. The error constant i is generated in the case where duplicate list values occur. Again, this bisemigroup was illustrated earlier. The `add_alpha_omega(`*bsg*`)` bisemigroup is similar to its semigroup counterparts, with the

sg	::=		(semigroup)
	|	`and`	(boolean conjunction)
	|	`or`	(boolean disjunction)
	|	`plus(`ty_num`)`	(integer addition)
	|	`plus_bound(`i`,`n`,`m`)`	(bounded integer addition)
	|	`times(`ty_num`)`	(integer multiplication)
	|	`times_bound(`i`,`n`,`m`)`	(bounded integer multiplication)
	|	`min(`ty_num_+`)`	(integer minimisation)
	|	`max(`ty_num_+`)`	(integer maximisation)
	|	`app(`ty`)`	(list append)
	|	`app_simp(`i`,`ty`)`	(simple list append)
	|	`union(`ty`)`	(set union)
	|	`inter(`ty`)`	(set intersection)
	|	`left(`ty`)`	(left)
	|	`right(`ty`)`	(right)
	|	`add_alpha(`i`,`sg`)`	(add alpha)
	|	`add_omega(`i`,`sg`)`	(add omega)
	|	`dir_prod(`i_1`:`sg_1`,...,`i_n`:`sg_n`)`	(direct product)
	|	`lex_prod(`i_1`:`sg_1`,...,`i_n`:`sg_n`)`	(lexicographic product)
	|	`disj_union(`i`,`i_1`:`sg_1`,...,`i_n`:`sg_n`)`	(disjoint union)
bsg	::=		(bisemigroup)
	|	`and_or`	(boolean conjunction/disjunction)
	|	`min_plus(`ty_num`)`	(integer minimisation/addition)
	|	`min_plus_bound(`i`,`n`,`m`)`	(bounded integer minimisation/addition)
	|	`min_times(`ty_num`)`	(integer minimisation/multiplication)
	|	`min_times_bound(`i`,`n`,`m`)`	(bounded integer minimisation/multiplication)
	|	`max_min(`ty_num`)`	(integer maximisation/minimisation)
	|	`union_inter(`ty`)`	(set union/intersection)
	|	`twin(`bsg`)`	(semigroup duplication)
	|	`swap(`bsg`)`	(semigroup swapping)
	|	`list_lex_app(`bsg`)`	(list lexicographic/append)
	|	`list_lex_app_simp(`i`,`bsg`)`	(bounded list lexicographic/append)
	|	`add_alpha_omega(`i`,`bsg`)`	(alpha/omega)
	|	`dir_prod(`i_1`:`bsg_1`,...,`i_n`:`bsg_n`)`	(direct product)
	|	`lex_prod(`i_1`:`bsg_1`,...,`i_n`:`bsg_n`)`	(lexicographic product)
	|	`disj_union(`i`,`i_1`:`bsg_1`,...,`i_n`:`bsg_n`)`	(disjoint union)

Figure 5.2: Syntax of RAML$_1$ semigroups and bisemigroups

exception that the constant i is an additive identity and multiplicative infinity. Finally, the `lex_prod(`i_1`:`bsg_1`,...,`i_n`:`bsg_n`)` bisemigroup corresponds to the n-ary bisemigroup lexicographic product.

5.3 Translation into IRL$_1$

In this section we describe how to translate from RAML$_1$ into IRL$_1$. For each syntactic category in RAML$_1$, we define a translation function $(\!|\_|\!)$ that maps into IRL$_1$. Recall from Chapter 4 that we can then use the semantic evaluation function $[\![\_]\!]$ to translate terms from IRL$_1$into the semantic domain D_1 to obtain their meaning. Hence the meaning of terms in RAML$_1$ is obtained by the composition $[\![\_]\!] \circ (\!|\_|\!)$.

Figure 5.3 gives translations of values and types, whilst Figures 5.4 and 5.5 give translations for semigroups and bisemigroups respectively. The definitions of the translations are straightforward, and therefore we do not further discuss the details.

We now return to the example from the beginning of the chapter:

```
lex_prod(dist : min_plus(int_non_neg),
         bw   : max_min(int_non_neg),
         path : list_lex_app_simp(NOTSIMP, min(int_pos)))
```

Upon translation into IRL$_1$, we obtain the following bisemigroup term:

```
(rec((dist, int_non_neg), (bw, int_non_neg),
      (path, add_const(NOTSIMP, list_simp(int_pos)))),
 lex_prod(min, max, list_lex(min)),
 dir_prod(plus, min, app))
```

Hence we see that the underlying type is a ternary record `rec(...)`. The additive operator is a lexicographic product `lex_prod(...)`, whilst the multiplicative operator is direct product `dir_prod(...)`. Each of these operators comprises three sub-operators, corresponding to the operations upon each of the individual record fields.

The example illustrates how the translation into IRL$_1$ exposes the computational content that is implicit in RAML$_1$ routing language specifications. From this stage it is now much easier to generate executable C++ code.

Type ty	Translation $(\!|ty|\!)$
`unit`	`unit`
`bool`	`bool`
`int`	`int`
`int_non_neg`	`int_non_neg`
`int_pos`	`int_pos`
`int_bound(`n`,`m`)`	`int_bound(`n`,`m`)`
`string`	`string`
`set(`ty`)`	`set(`$(\!|ty|\!)$`)`
`list(`ty`)`	`list(`$(\!|ty|\!)$`)`
`list_simp(`ty`)`	`list_simp(`$(\!|ty|\!)$`)`
`rec(`i_1`:`ty_1`,...,`i_n`:`ty_n`)`	`rec((`i_1`,`$(\!|ty_1|\!)$`),...,(`i_n`,`$(\!|ty_n|\!)$`))`
`disj_union(`i_1`:`ty_1`,...,`i_n`:`ty_n`)`	`disj_union((`i_1`,`$(\!|ty_1|\!)$`),...,(`i_n`,`$(\!|ty_n|\!)$`))`
`add_const(`i`,`ty`)`	`add_const(`i`,`$(\!|ty|\!)$`)`

Value v	Translation $(\!|v|\!)$
`unit`	`unit`
`true`	`true`
`false`	`false`
n	`int(`n`)`
s	`string(`s`)`
`[`v_1`,...,`v_n`]`	`[`$(\!|v_1|\!)$`,...,`$(\!|v_n|\!)$`]`
`{`v_1`,...,`v_n`}`	`{`$(\!|v_1|\!)$`,...,`$(\!|v_n|\!)$`}`
`<`i_1`=`v_1`,...,`i_n`=`v_n`>`	`(`$(\!|v_1|\!)$`,...,`$(\!|v_n|\!)$`)`
`inj(`i`, `v`)`	`inj(`i`, `$(\!|v|\!)$`)`
i	`const(`i`)`

Figure 5.3: Translations of RAML$_1$ types and values into IRL$_1$

Semigroup sg	Translation $(\!|sg|\!)$
`and`	`(bool, and)`
`or`	`(bool, or)`
`plus(`ty_num`)`	`(`$(\!|ty\_num|\!)$`, plus)`
`plus_bound(`i`,`n`,`m`)`	`(add_const(`i`,int_bound(`n`,`m`)), plus)`
`times(`ty_num`)`	`(`$(\!|ty\_num|\!)$`, times)`
`times_bound(`i`,`n`,`m`)`	`(add_const(`i`,int_bound(`n`,`m`)), times)`
`min(`ty_num_+`)`	`(`$(\!|ty\_num_+|\!)$`, min)`
`max(`ty_num_+`)`	`(`$(\!|ty\_num_+|\!)$`, max)`
`app(`ty`)`	`(list(`$(\!|ty|\!)$`), app)`
`app_simp(`i`,`ty`)`	`(add_const(`i`,list_simp(`$(\!|ty|\!)$`)), app)`
`union(`ty`)`	`(set(`$(\!|ty|\!)$`), union)`
`inter(`ty`)`	`(set(`$(\!|ty|\!)$`), inter)`
`left(`ty`)`	`(`$(\!|ty|\!)$`, left)`
`right(`ty`)`	`(`$(\!|ty|\!)$`, right)`
`add_alpha(`i`,`sg`)`	`(add_const(`i`,`ty`), alpha(`bo`))`, where `(`ty`, `bo`) = `$(\!|sg|\!)$
`add_omega(`i`,`sg`)`	`(add_const(`i`,`ty`), omega(`bo`))`, where `(`ty`, `bo`) = `$(\!|sg|\!)$
`dir_prod(`i_1`:`sg_1`,...,`i_n`:`sg_n`)`	`(rec((`i_1`,`ty_1`),...,(`i_n`,`ty_n`)), dir_prod(`bo_1`,...,`bo_n`))`, where `(`ty_i`, `bo_i`) = `$(\!|sg_i|\!)$
`lex_prod(`i_1`:`sg_1`,...,`i_n`:`sg_n`)`	`(rec((`i_1`,`ty_1`),...,(`i_n`,`ty_n`)), lex_prod(`bo_1`,...,`bo_n`))`, where `(`ty_i`, `bo_i`) = `$(\!|sg_i|\!)$
`disj_union(`i`,`i_1`:`sg_1`,...,`i_n`:`sg_n`)`	`(add_const(`i`,disj_union((`i_1`,`ty_1`),...,(`i_n`,`ty_n`))), disj_union(`bo_1`,...,`bo_n`))`, where `(`ty_i`, `bo_i`) = `$(\!|sg_i|\!)$

Figure 5.4: Translations of RAML$_1$ semigroups into IRL$_1$

Bisemigroup *bsg*	Translation $(\!|bsg|\!)$
`and_or`	`(bool, and, or)`
`min_plus(`ty_num`)`	$(\!|ty\_num|\!)$`, min, plus)`
`min_plus_bound(`i`,`n`,`m`)`	`(add_const(`i`,int_bound(`n`,`m`)), min, plus)`
`min_times(`ty_num`)`	$(\!|ty\_num|\!)$`, min, times)`
`min_times_bound(`i`,`n`,`m`)`	`(add_const(`i`,int_bound(`n`,`m`)), min, times)`
`max_min(`ty_num_+`)`	$(\!|ty\_num_+|\!)$`, max, min)`
`union_inter(`ty`)`	`(set(`$(\!|ty|\!)$`)), union, inter)`
`twin(`sg`)`	(ty, bo, bo), where $(ty, bo) = (\!|sg|\!)$
`swap(`bsg`)`	(ty, bo_2, bo_1), where $(ty, bo_1, bo_2) = (\!|bsg|\!)$
`list_lex_app(`sg`)`	`(list(`$(\!|ty|\!)$`)), list_lex(`bo`), app),` where $(ty, bo) = (\!|sg|\!)$
`list_lex_app_simp(`i`,`sg`)`	`(add_const(`i`,list_simp(`ty`)),` `list_lex(`bo`), app),` where $(ty, bo) = (\!|sg|\!)$
`add_alpha_omega(`i`,`bsg`)`	`(add_const(`i`,`ty`), alpha(`bo_1`), omega(`bo_2`)),` where $(ty, bo_1, bo_2) = (\!|bsg|\!)$

`dir_prod(`i_1`:`bsg_1`,...,`i_n`:`bsg_n`)`	`(rec((`i_1`,`ty_1`),...,(`i_n`,`ty_n`)),`
	`dir_prod(`$bo_{1,1}$`,...,`$bo_{n,1}$`),`
	`dir_prod(`$bo_{1,2}$`,...,`$bo_{n,2}$`)),`
	where $(ty_i, bo_{i,1}, bo_{i,2}) = (\!|bsg_i|\!)$

`lex_prod(`i_1`:`bsg_1`,...,`i_n`:`bsg_n`)`	`(rec((`i_1`,`ty_1`),...,(`i_n`,`ty_n`)),`
	`lex_prod(`$bo_{1,1}$`,...,`$bo_{n,1}$`),`
	`dir_prod(`$bo_{1,2}$`,...,`$bo_{n,2}$`)),`
	where $(ty_i, bo_{i,1}, bo_{i,2}) = (\!|bsg_i|\!)$

`disj_union(`i`,`i_1`:`bsg_1`,...,`i_n`:`bsg_n`)`	`(add_const(`i`,disj_union((`i_1`,`ty_1`),...,(`i_n`,`ty_n`))),`
	`disj_union(`$bo_{1,1}$`,...,`$bo_{n,1}$`),`
	`disj_union(`$bo_{1,2}$`,...,`$bo_{n,2}$`)),`
	where $(ty_i, bo_{i,1}, bo_{i,2}) = (\!|bsg_i|\!)$

Figure 5.5: Translations of RAML$_1$ bisemigroups into IRL$_1$

Chapter 6

Compilation

In Chapter 5, we described how to translate RAML_1 routing language specifications into IRL_1 terms. In this chapter we describe compilation-proper, whereby IRL_1 terms are translated into executable code. Whilst both steps can be seen as a form of compilation, we prefer to reserve the term for this last stage where actual low-level code is generated. We commence by giving an overview of our approach to compilation (§ 6.1), before describing the specifics of the compilation process (§ 6.2).

6.1 Overview

Recall the example RAML_1 specification from Chapter 5:

```
lex_prod(dist : min_plus(int_non_neg),
         bw   : max_min(int_non_neg),
         path : list_lex_app_simp(NOTSIMP, min(int_pos)))
```

We have already demonstrated how to translate such RAML_1 specifications into our intermediate language, IRL_1. For example, we have shown that the preceding specification is translated into the following IRL_1 term:

```
(rec((dist, int_non_neg), (bw, int_non_neg),
      (path, add_const(NOTSIMP, list_simp(int_pos)))),
 lex_prod(min, max, list_lex(min)),
 dir_prod(plus, min, app))
```

We now describe how to translate such IRL_1 terms into efficient, executable code that implements a specified routing interface.

Compilation of an IRL_1 term involves mapping its components onto templated C++ code (we discuss template metaprogramming in Section 6.1.1). This code is then

further compiled into machine code by a C++ compiler. The templated code is actually contained within an external template library, and thus the majority of the effort simply involves selecting appropriate datatypes and functions. Effectively, we have used the template facilities of C++ to embed IRL_1 within a library. By targeting C++ we also benefit from the availability of mature, optimising compilers. This frees us from the concerns of low-level optimisations. Furthermore, the language contains the Standard Template Library (STL), which contains many useful datatypes such as lists and sets. We reuse these to reduce the size of the external library. Finally, the routing algorithms (§ 2.7) are themselves written in C/C++, and therefore it is straightforward to interface with this code.

This contrasts with our original approach that involved generating C code by embedding it as strings within the compiler itself. Compilation then amounted to emitting these strings, with names substituted as necessary. This technique caused the source code for the compiler to become very large due to the many thousands of lines of embedded C code. Furthermore, there was no assurance of type safety for the embedded code because it was represented as opaque strings within the compiler. Therefore it was only possible to test the embedded code by emitting it and passing it through a C compiler. In Section 6.1.1 we explain why it is necessary to adopt C++, instead of C, so that this embedded code can be abstracted out into an external library.

An IRL_1 term comprises types and operators. Here we give an overview of their translation into C++. We leave a full description to Sections 6.2.1 and 6.2.2 respectively. Firstly, each IRL_1 type is mapped onto the name of a C++ class. Returning to our example, the IRL_1 type

```
rec((dist, int_non_neg), (bw, int_non_neg),
     (path, add_const(NOTSIMP, list_simp(int_pos)))
```

is translated into the C++ class name

```
RecWrap<
  RecCons<dist, IntNonNeg,
  RecCons<bw, IntNonNeg,
  RecCons<path, AddConst<NOTSIMP, ListSimp<IntPos> >,
  RecLast> > > >
```

The names within angle brackets are in fact *template parameters*; we introduce templates in Section 6.1.1. Also, identifiers such as `dist`, `bw` and `path` correspond to string constants with the values `"dist"`, `"bw"` and `"path"` respectively. We discuss the definitions of the record classes `RecWrap`, `RecCons` and `RecLast` in Section 6.2.3. Each such class contains methods to interconvert between the internal format, a wire format (for distributed algorithms), and a textual format (for parsing and printing configuration).

The second constituent of IRL_1 terms are operators. Each such operator is mapped onto a functor. This is a class that overloads the function call operator `()`, allowing objects of the class to be invoked as if they are functions. Again returning to our example, the IRL_1 operator

```
lex_prod(min, max, list_lex(min))
```

is translated into the C++ functor name

```
RecOpLexWrap<
  RecOpLexCons<0, IntNonNeg, IntMin,
  RecOpLexCons<1, IntNonNeg, IntMax,
  RecOpLexCons<2, ListSimp<IntPos>, ListSimpOpLex<IntMin>,
  RecOpLexLast> > > >
```

Again, Section 6.2.3 gives more details about the representation of records. Finally, the code is wrapped so that it implements the specified routing interface. We diagrammatically represent this process at the end of this chapter in Figure 6.5.

6.1.1 Template metaprogramming

C++ templates are a language feature for abstracting over types. Types and constants are substituted into templated classes and functions to create regular versions of these structures. Templates add significant flexibility to the C++ language, increasing code reusability and reducing the need for code duplication. Moreover, because substitution of types occurs at *compile-time*, it is possible to generate efficient code using techniques such as function inlining. In fact, templates permit a Turing-powerful form of offline partial evaluation [100, 101]. It is for this reason that programming using templates is often referred to as *metaprogramming*; the programmer is writing code that itself generates code. This is particularly useful for our application.

The power of C++ template metaprogramming has already been exploited by several linear algebra libraries. For example, Eigen [102] supports templated vectors and matrices. The template metaprogramming is used for optimisation techniques such as loop unrolling, vectorisation and the elision of dynamic memory allocation. Blitz++ [103] is another C++ library for linear algebra that again exploits template metaprogramming to increase performance. The Matrix Template Library [104] uses template metaprogramming for efficiently changing the representation of matrices.

We illustrate template metaprogramming using the example of the 'set-plus' operator

$$xs \oplus ys = \{x \oplus' y \mid x \in xs \wedge y \in ys\}.$$

The operator combines elements of the sets *xs* and *ys* using some additional operator $\oplus'$. Note this operator is used in the `minset_union_plus` translation in IRL_2. We further discuss the implementation of operators in Section 6.2.2.

We first consider implementing this operator as a C function. In order to implement this operator within a library, it is necessary to abstract over the operator $\oplus'$. Within C, this requires parameterising the function upon a function pointer. That is, for some set type `SetT`, we obtain the following function signature for the implementation of the set-plus operator:

```
SetT SetPlus(void *(*op)(const void *xs, const void *ys),
             xs SetT,
             ys SetT);
```

The first parameter, `op`, is a function pointer corresponding to the implementation of $\oplus'$, and the second and third parameters are the set arguments *xs* and *ys*.

Whilst this approach will indeed work, there remain two significant disadvantages due to the weak abstraction facilities provided by the C language. Firstly, there are runtime costs associated with function pointers. These costs manifest themselves both when passing function pointers around as additional parameters and also when invoking them (unless the compiler can perform function inlining). Secondly, by resorting to void pointers, we lose type-safety; this was one of the reasons for moving to a library in the first place.

We now consider implementing the SetPlus operator in the C++ language. The C++ template facilities allow us to easily create multiple copies of functions such as `SetPlus` *at C++ compile time*, with each such copy specialised to a particular `op`. This eliminates the additional `op` parameter, and also permits optimisation techniques such as the inlining of the `op` function. Effectively, the C++ compiler is performing much of the work that was originally occurring in the 'embedded C' version of the compiler. We also have the additional benefits of type-safety and increased maintainability.

Figure 6.1 illustrates the template metaprogramming technique using the C++ implementation of the `SetPlus` function. This code is implemented as a functor i.e. it contains an `operator()` method. The parameter `op` is now a functor that is substituted into the body of the structure at compile-time. The type parameter `T` is automatically inferred by the C++ compiler, and corresponds to the type of the set elements. The actual application of `op` occurs in the command

```
res.data_.insert(op()(*x, *y));
```

Here, `op` is instantiated and applied to a pair of elements. The returned value is then inserted into the result set. The set-plus functor is itself applied to pairs of arguments

```
template <typename op>
struct SetPlus
{
  template<typename T>
  Set<T> operator()(const Set<T>& xs, const Set<T>& ys) const
  {
    Set<T> res;
    typename std::set<T>::const_iterator x, y;

    for (x = xs.data_.begin(); x != xs.data_.end(); x++)
      for (y = ys.data_.begin(); y != ys.data_.end(); y++)
        res.data_.insert(op()(*x, *y));

    return res;
  }
};
```

Figure 6.1: Templated set-plus functor

as `SetPlus<op>()(xs, ys)`. For each distinct `op`, a different copy of `SetPlus` is created by the compiler.

6.2 Compilation

The compilation process is inductively defined over the structure of IRL_1 terms. For each syntactic category within IRL_1, we define a translation into a C++ class. It is also necessary to perform some minimal book-keeping such as generating fresh names for types and tracking which library files to include. Much of this is standard, and therefore we do not further elaborate upon the details. We now discuss the details of the compilation of types (§ 6.2.1) and operators (§ 6.2.2).

6.2.1 Types

Recall that IRL_1 types are represented as C++ classes. In many cases these classes are wrappers around C++ Standard Template Library (STL) types, such as vectors and sets. In the case of unbound integers, we use the GNU Multiple Precision Arithmetic library (GMP) [105]. This supports arbitrary precision integer arithmetic, and means that we are not limited by the C++ fixed-precision integer types.

Figure 6.2 gives the definition of the `Set<T>` class, which is used to represent the IRL_1 set type. We use this example to describe the general interface implemented by all

```
template <typename T>
class Set
{
public:
  Set();
  Set(const std::set<T>& data);
  Set(const struct ast_sig_t& ast);
  Set(char **buf, size_t *size);
  static Set rand();

  bool operator==(const Set& x) const;
  bool operator!=(const Set& x) const;
  bool operator<(const Set& x) const;

  friend std::ostream&
    operator<<(std::ostream& os, const Set& xs) { /*... */ }

  void marsh(char **buf, size_t *size, size_t *total) const;
  uint32_t hash() const;

  std::set<T> data_;
};
```

Figure 6.2: Templated set class, used to represent the IRL_1 set type.

classes that represent IRL_1 types. In this example, the parameter `T` corresponds to the representation of the element type. Many of following functions that we describe, such as parsing and printing, will recursively invoke functions upon `T` for this example.

Parsing Configurations are converted to abstract syntax trees (ASTs), using a generic, YACC-based parser [64]. The parser is not specialised to any particular routing language, but instead is capable of parsing all value types. Each class then has a constructor which accepts such ASTs, represented using the `ast_sig_t` type. The class raises an error if the AST does not represent a value of the correct type, or otherwise creates an object that corresponds to the concrete value.

Printing Each class extends the C++ stream-based printing mechanism by overloading `operator<<`. This operator is invoked whenever a textual representation of a value is needed e.g. for debugging, or for observing routing solutions. Note that values that are parsed and then printed should be semantically identical to their original, textual representation (the order of set elements may be permuted, for example).

Marshalling Each class implements a marshalling function `marsh` to convert objects to byte-strings for network transmission. For efficiency reasons, each marshalling

function directly appends to the specified buffer, `buf`. This avoids performance loss due to copying. The marshal representations of variable length values, such as sets, are prefixed with their length. This allows values to be locally unmarshalled, instead of requiring reference to some index of value offsets and lengths.

Unmarshalling Each class has a constructor that is used for unmarshalling values. This constructor accepts byte-strings and attempts to validate them. If it succeeds, then an object of the correct value is instantiated, otherwise an error is raised.

Random instances The `rand` method is used for creating pseudo-random instances of classes. This functionality is used to dynamically create example matrices for the offline algorithms.

Ordering The operator < defines a total order over instances of the class. That is, given any pair of non-identical objects, we may say that one is strictly more preferred than the other. This is necessary so that values can be inserted into STL container types such as sets. The operators == and `!=` are consistent with this ordering. We note that such orders are distinct from the orders that we add to the metalanguage in Chapter 7.

Hashing The `hash` function is used to obtain a hashed representation of objects. This is used both for placing objects into STL hash tables, and also within Quagga to ensure that each unmarshalled metric is stored just once.

6.2.2 Operators

Recall from Section 6.1 that IRL_1 operators are represented as C++ functors. We illustrated the functor representation of the set-plus operator in Figure 6.1. This example demonstrates how functor definitions can be combined to produce new functors, which is similar to the manner in which classes representing IRL_1types are combined. In general, the functor representations of operators are relatively straightforward. In many cases we are able to reuse STL functors, such as for the set union operation.

6.2.3 Records and unions

In this section we discuss our implementation of records and unions, each of which is based upon a template metaprogramming technique known as *typelists* [106]. We first introduce typelists, before describing how we have applied it to our language implementation.

Typelists allow arbitrary collections of types; we will use the construct to represent the different types in records and unions. A typelist is simply defined as a structure that takes two type parameters. We give the definition of a typelist in Figure 6.3(a). Typelists can be chained together by instantiating the second typelist parameter as

```
template <class T, class U>
struct Typelist
{
  typedef T Head;
  typedef U Tail;
};

class NullType {};
```

(a) Definition of a typelist

```
template <class TList> struct Length;
template <> struct Length<NullType>
{
  enum { value = 0 };
};
template <class T, class U>
struct Length
{
  enum { value = 1 + Length<U>::value };
};
```

(b) Static length computation

Figure 6.3: Typelist definitions and operations

another typelist. For example, we define a typelist `MyTy` containing an integer and a character as

```
typedef Typelist<int, Typelist<char, NullType> > MyTy;
```

Here, `NullType`, again defined in Figure 6.3(a), is simply used as a distinguished type to denote the end of the list.

Typelists admit compile-time computations because they are defined at the template-level. For example, in Figure 6.3(b), we give the definition of a structure to statically compute the length of a typelist. The length of `NullType` is defined as 0, whilst the length of `Typelist<T, U>` is recursively defined as the length of `U` plus one. For example, `Length<MyTy>::value` evaluates at compile-time to the constant 2. Note that both of the code samples in Figure 6.3 are taken from [106].

We now describe how we extend typelists to define record types (union types are defined similarly). The main class involved in our implementation of record types is `RecCons<S, T, U>`. This class can be seen as a generalisation of `Typelist`. The parameter `S` is a string specifying the record field name, whilst the parameter `T` is a class corresponding to the field type. The final parameter, `U`, is either a `RecCons` or a `RecLast`, and represents the remainder of the record. Recall that the example RAML_1 record type

```
rec((dist, int_non_neg), (bw, int_non_neg),
     (path, add_const(NOTSIMP, list_simp(int_pos))))
```

is represented as

```
RecWrap<
  RecCons<dist, IntNonNeg,
```

```
template <const std::string& S, typename T, typename U>
class RecCons
{
  // Constructors (some omitted)
  RecCons(char **buf, size_t *size) { /* ... */ }
  RecCons(const T& value, const U& next) : value_(value), next_(next) { }

  // Operators (some omitted)
  bool operator==(const RecCons& x) const
  {
    return value_ == x.value_ && next_ == x.next_;
  }

  T value_;
  U next_;
};
```

Figure 6.4: Abbreviated definition of `RecCons`

```
RecCons<bw, IntNonNeg,
RecCons<path, AddConst<NOTSIMP, ListSimp<IntPos> >,
RecLast> > > >
```

The outer type, `RecWrap`, simply provides a clean interface to the the rest of the record, whilst `RecLast` is roughly analogous to `NullType`. We give an outline of the definition of `RecCons<S, T, U>` in Figure 6.4. Note how the equality operator is defined in terms of the equality operators over `T` and `U`. We also use a typelist-based approach to define types and operators over records and unions.

The typelist-based approach has the advantages that it both integrates well with our template-based approach to compilation, and also maintains static safety. Furthermore, due to the template metaprogramming, much of the code is 'compiled-away' by the C++ compiler. An alternative approach would be to programmatically generate a class definition for each different record size. This technique leads to large class sizes, with correspondingly slower C++ compilation times. Furthermore, maintenance of code becomes more difficult. We therefore believe that our typelist-based approach is better-suited to our application.

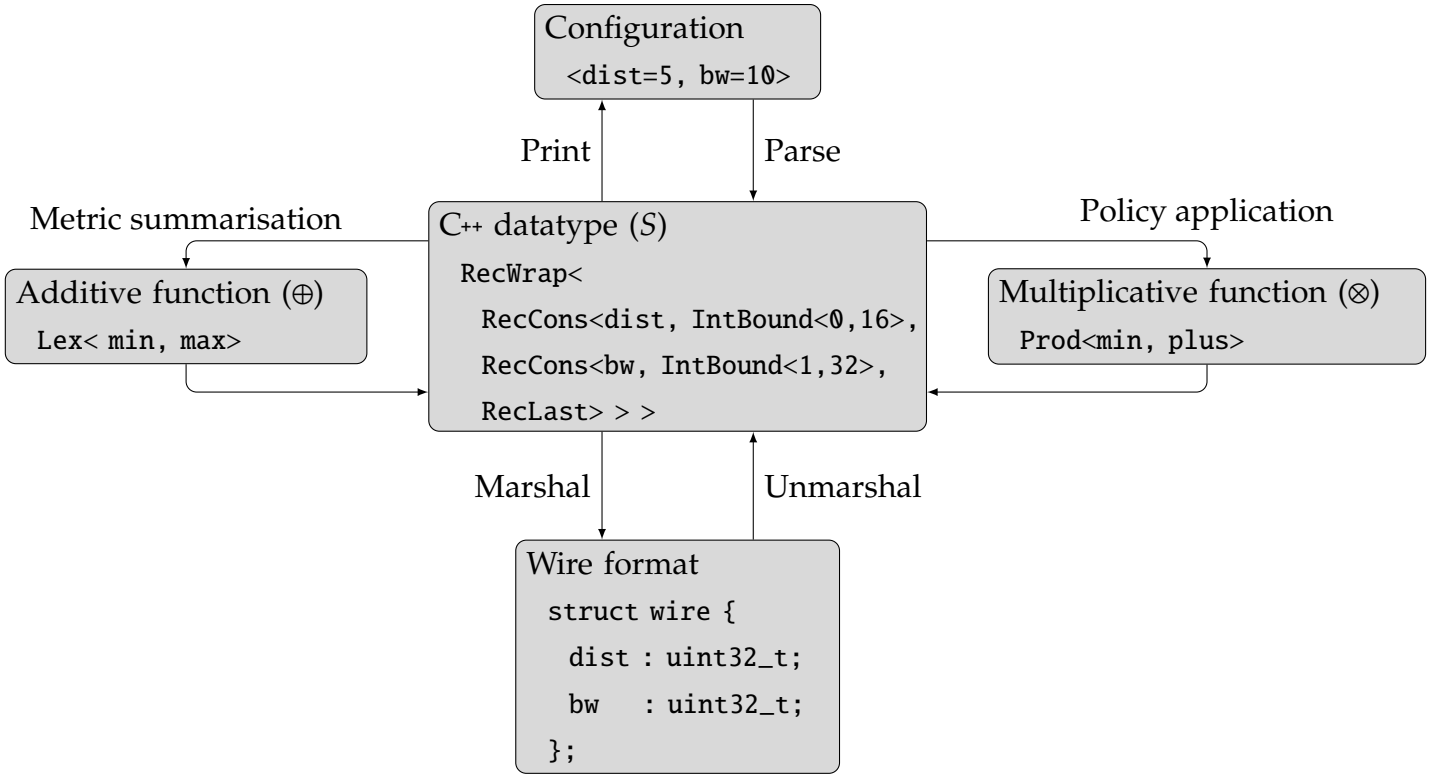


Figure 6.5: Compilation of IRL_1 bisemigroup into a datatype and additive and multiplicative functions over that datatype. We also show utility functions for inter-converting the datatype into textual configuration and a wire format for transfer between routing daemons.

Chapter 7

RAML_2: Extended metalanguage

In this chapter we show how to extend RAML_1 to increase its expressivity. We call the extended metalanguage RAML_2. We commence by motivating the adoption of a more expressive metalanguage (§ 7.1). Next we define the extended semantic domain in which we model RAML_2 (§ 7.2). In particular, we add *orders* and *transforms* to the domain. Finally, we describe the metalanguage (§ 7.3), and sketch how it is translated into the intermediate language IRL_2.

7.1 Examples

In this section we describe two routing languages that cannot be expressed within RAML_1. This motivates our adoption of a more expressive metalanguage. We informally define the examples using RAML_2, explaining the language features as we go along. The definition of RAML_2 will be made precise in Section 7.3.

7.1.1 Sets of minimal-length paths

We first address the problem that occurs when there are multiple equivalent or incomparable paths between nodes. For example, suppose that metrics comprise simple lists of arc identifiers, with shorter lists preferred. This has a very rough similarity to AS paths. We then face the difficulty that given two distinct paths p and q, both may be of the same length. How might we summarise these paths? We illustrate the problem in Figure 7.1. Here there are two minimal-length paths between nodes a and e: $[(a, b), (b, e)]$ and $[(a, c), (c, e)]$.

The solution is to represent metrics as *sets* of minimal-length paths. For example, the metric between a and e could be represented as $\{[(a, b), (b, e)], [(a, c), (c, e)]\}$. No list within such a set is 'smaller' than another; we term such structures *minimal sets*. Policy

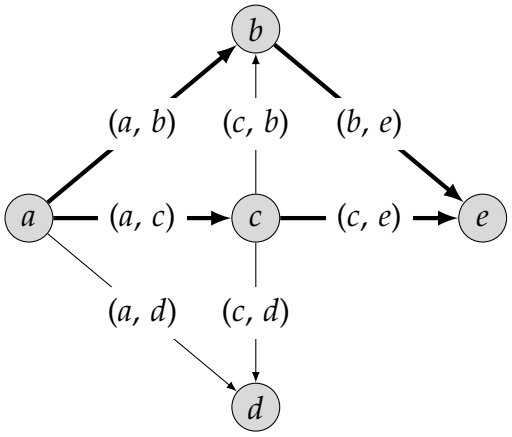


Figure 7.1: Minimal-length paths example. Path weights comprise lists of arc identifiers, with shorter lists preferred. The two minimal-length paths between nodes *a* and *e* are [(*a*, *b*), (*b*, *e*)] and [(*a*, *c*), (*c*, *e*)]. Both of these paths are denoted in bold. These two paths can be represented as the *minimal set* {[(*a*, *b*), (*b*, *e*)], [(*a*, *c*), (*c*, *e*)]}.

is applied to sets element-wise, with sets summarised using the set-union operation. The resulting sets then must be *minimised* to remove any non-minimal lists.

There are two steps to modelling this routing language in RAML$_2$. Firstly, we define the `paths` sub-language as

```
let paths : order_transform = list_simp_lte_cons(NOTSIMP, string)
```

The `order_transform` annotation denotes the kind of the algebra, which we further describe in Section 7.2. Metrics within the language comprise simple lists of strings, with each string denoting an arc. An example of a metric value is `["(a, b)", "(b, e)"]`. Invalid paths are represented as the constant `NOTSIMP`. Shorter lists are preferred, with `NOTSIMP` least preferred. Policy is represented as strings, such as `"(b, e)"`, which are 'consed' onto the head of lists. Note that RAML$_1$ does not support different types for metric and policy.

The second step in modelling the routing language is to apply a minimal sets construction to the `paths` language:

```
let min_paths : semigroup_transform = minset_union_map(paths)
```

We properly define minimal sets in Section 7.2.3, noting here that it is not possible to represent minimal sets within RAML$_1$. Metrics within the `min_paths` routing language comprise sets of minimal-length lists of strings. For example, the set `{["(a, b)", "(b, e)"], ["(a, c)", "(c, e)"]}` is a metric. Policy remains represented as strings, such as `"(b, e)"`. Sets are summarised using the union operator, with lists that are not of minimal-length removed. Policy is applied by invoking the 'cons' operator on every list within a set, again with non-minimal lists then removed.

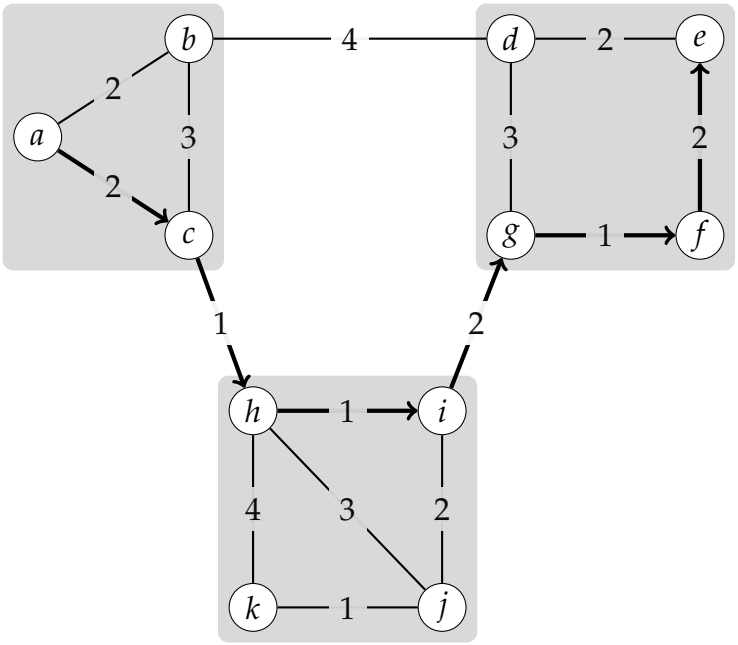


Figure 7.2: Administrative regions example. Shaded boxes indicate administrative regions and arc labels indicate distances. Paths between regions are selected according to *inter-region* distances only, allowing the administrators of each region to independently configure *intra-region* distances. Bold arcs indicate the selected path from a to e.

7.1.2 Administrative regions

Recall from Chapter 1 that BGP is being used as an IGP in order to allow increased administrative delegation inside large, geographically distributed networks. In this section we describe a routing language that allows hierarchical control of path selection; this example can be seen as a first step towards a 'BGP-like' IGP.

Consider the situation shown in Figure 7.2, where there are three administrative regions. Shaded boxes denote regions and arc weights denote distances. Our desired routing model is for paths to be selected according to *inter-region* distances, with *intra-region* distances only used to locally select paths within regions. Changing distances within a region should not affect inter-region paths. This allows the administrator of each region to independently configure internal policy without affecting inter-region traffic flows.

The bold path in Figure 7.2 is selected according to the routing language informally described above. The path has an inter-region distance of 3. The 'upper' inter-region path, which traverses arc (b, d), is *not* selected. This is because choosing this path would result in a greater inter-region distance of 4. Intra-region paths are then selected to minimise the distance within each region. For example, the intra-region path $\langle a, c \rangle$ is chosen over $\langle a, b, c \rangle$ because the former has an intra-region distance of 2, whereas the latter has a greater intra-region distance of 5.

```
let lte_plus : order_transform =
  cayley_left (order_left(min_plus_pos))

let inter_region : order_transform =
  lex_prod(
    edist : lte_plus,
    epath : paths,
    idist : const(lte_plus),
    ipath : const(paths))

let intra_region : order_transform =
  lex_prod(
    edist : id(lte_plus),
    epath : id(paths),
    idist : lte_plus,
    ipath : paths)

let regions : order_transform =
  disj_union(ERROR,
    external : inter_region,
    internal : intra_region)
```

Figure 7.3: Routing language for modelling administrative regions

Figure 7.3 defines an implementation of the routing language in RAML$_2$. The `lte_plus` sub-language defines the conventional $\leq$ order over the integers, whilst `paths` is as defined in the previous example. The `inter_region` and `intra_region` sub-languages are respectively used to model inter-region and intra-region policy. Both sub-languages have identical metrics of the form

<edist=m, epath=p, idist=n, ipath=q>

where m and p are external distances and paths, and n and q are internal distances and paths respectively. Our use of a single metric type means that both inter-region and intra-region policy may operate upon a given metric. An alternative design would be to have distinct inter-region and intra-region metric types, although this would necessitate the additional complexity of defining functions for mapping between the two types.

The `regions` routing language specifies that inter-region policy is tagged with the keyword `external`, leading to policy of the form

inj(external, <edist=m, epath=p, idist=n, ipath=q>)

This policy causes the sub-policy m and p to be applied to the external distance and path components, whilst the internal distance and path are reset to the values n and q respectively. The latter behaviour is specified by the `const` keyword. Resetting internal distances and paths ensures that these values do not 'leak' between regions, compromising the independent configuration of intra-region policy.

Returning to the `regions` routing language, intra-region policy is tagged with the keyword `internal`, resulting in policy of the form

```
inj(internal, <edist=m, epath=p, idist=n, ipath=q>)
```

This causes the sub-policy n and q to be applied to the internal distance and path components. The external distance and path policy are ignored, with the corresponding metric fields simply copied. The behaviour is specified by the `id` keyword (we have developed a language extension that can automatically elide these fields, which would cause internal policy to only contain `idist` and `ipath` fields).

This example cannot be expressed in RAML$_1$ because the language does not support a functional model of policy application. This facility is required for representing the `const` and `id` operations.

7.2 Semantic domain

The RAML$_1$ metalanguage is based upon semigroups and bisemigroups. In this section we show how bisemigroups can be seen as a specific instance of a larger class of algebraic structures that may be used for modelling routing languages. This generalised set of algebraic structures forms the semantic basis of RAML$_2$. More complete descriptions of these structures can be found in [91].

We first consider the task of weight summarisation. *Algebraic* summarisation uses a semigroup $(S, \oplus)$. Weights $s_1, s_2 \in S$ are summarised as $s_1 \oplus s_2 \in S$. This is the method that we introduced in Chapter 2 and used for defining RAML$_1$ in Chapter 5. An alternative is to instead use *ordered* summarisation. This makes use of a *order* $(S, \preccurlyeq)$. We say that s_1 is *more preferred* or *less than* s_2 if and only if $s_1 \preccurlyeq s_2$.

We now consider policy. *Algebraic* policy uses a semigroup $(S, \oplus)$. Policy $s_1 \in S$ is applied to a metric $s_2 \in S$ as $s_1 \otimes s_2 \in S$. Again, we introduced this type of policy in Chapter 2 and used it for defining RAML$_1$ in Chapter 5. An alternative approach is to use *functional* policy. Here, we have a set of functions $F \subseteq S \to S$. Policy $f \in F$ is applied to a metric $s \in S$ as the function application $f(s)$.

Combining the two summarisation methods with the two forms of policy we obtain four different kinds of algebras. These are illustrated in Figure 7.4. In many cases it

Weight computation	Weight summarisation: **Algebraic**	Weight summarisation: **Ordered**
Algebraic	Bisemigroups $(S, \oplus, \otimes)$ Semirings [4, 5, 88] Non-distributive semirings [107, 108]	Order Semigroups $(S, \preccurlyeq, \otimes)$ Ordered semirings [109, 110, 111] QoS algebras [112]
Functional	Semigroup Transforms $(S, \oplus, F)$ Monoid endomorphisms [5, 88]	Order Transforms $(S, \preccurlyeq, F)$ Sobrinho structures [3, 113]

Figure 7.4: The quadrants model of algebraic routing, from [91]

is possible to translate between the various kinds of algebras, depending upon their mathematical properties. In this chapter we show how to expand the metalanguage to use all four kinds of structures. We now discuss the different kinds of algebra.

7.2.1 Orders

An order $(S, \preccurlyeq)$ is a binary relation $\preccurlyeq$ over a set S. Orders denoted by routing languages are normally *preorders*. A preorder is a *reflexive* and *transitive* order. An order is reflexive if for all $s \in S$,

$$s \preccurlyeq s \quad (\textsc{refl})$$

and transitive if for all $s_1, s_2, s_3 \in S$,

$$s_1 \preccurlyeq s_2 \wedge s_2 \preccurlyeq s_3 \Rightarrow s_1 \preccurlyeq s_3 \quad (\textsc{trans}).$$

The reflexivity and transitivity properties ensure that metrics summarisation has the intuitive semantics that we might expect. We define a selection of algebraic properties of orders in Figure 7.5. We note that a *partial order* is an anti-symmetric preorder whilst an *equivalence relation* is a symmetric preorder.

Given an order $(S, \preccurlyeq)$, we can define the following relations:

$$\begin{aligned} s_1 \prec s_2 &\iff (s_1 \preccurlyeq s_2) \wedge (s_2 \not\preccurlyeq s_1) \\ s_1 \sim s_2 &\iff (s_1 \preccurlyeq s_2) \wedge (s_2 \preccurlyeq s_1) \\ s_1 \sharp s_2 &\iff (s_1 \not\preccurlyeq s_2) \wedge (s_2 \not\preccurlyeq s_1) \end{aligned}$$

For $s_1, s_2 \in S$, we say that s_1 is *strictly less than* s_2 if $s_1 \prec s_2$, *equivalent* if $s_1 \sim s_2$ and *incomparable* if $s_1 \sharp s_2$.

Property	Definition	Preorder	Partial order	Equivalence relation
REFL	$x \preccurlyeq x$	✓	✓	✓
TRANS	$x \preccurlyeq y \land y \preccurlyeq z \Rightarrow x \preccurlyeq z$	✓	✓	✓
SYM	$x \preccurlyeq y \Rightarrow y \preccurlyeq x$		×	✓
ANTISYM	$x \preccurlyeq y \Rightarrow x = y$		✓	×
TOTAL	$x \preccurlyeq y \lor y \preccurlyeq x$			
TOP	$\exists \alpha_{\preccurlyeq} \in S.\, x \preccurlyeq \alpha_{\preccurlyeq}$			
BOT	$\exists \omega_{\preccurlyeq} \in S.\, \omega_{\preccurlyeq} \preccurlyeq x$			

Figure 7.5: Properties of orders $(S, \preccurlyeq)$. All free variables are universally quantified.

Now suppose that we have a bisemigroup $(S, \oplus)$. We can translate it into an order $(S, \preccurlyeq)$ in two ways:

$$\begin{aligned} \text{order\_left}(S, \oplus) &= (S, \preccurlyeq_L) \quad \text{where } s_1 \preccurlyeq_L s_2 \iff s_1 = s_1 \oplus s_2 \\ \text{order\_right}(S, \oplus) &= (S, \preccurlyeq_R) \quad \text{where } s_1 \preccurlyeq_R s_2 \iff s_2 = s_1 \oplus s_2 \end{aligned}$$

We refer to the resulting orders as the *left* and *right* translations, respectively. Similarly, we can also translate a total preorder into a selective semigroup:

$$\text{semigroup\_left}(S, \preccurlyeq) = (S, \oplus_L) \quad \text{where } s_1 \oplus_L s_2 = \begin{cases} s_1 & \text{if } s_1 \preccurlyeq s_2 \\ s_2 & \text{otherwise} \end{cases}$$

$$\text{semigroup\_right}(S, \preccurlyeq) = (S, \oplus_R) \quad \text{where } s_1 \oplus_R s_2 = \begin{cases} s_2 & \text{if } s_1 \preccurlyeq s_2 \\ s_1 & \text{otherwise} \end{cases}$$

We can use these translations to convert between algebraic and ordered weight summarisation. We diagrammatically represent these translations and the following translations in Figure 7.6.

Note that the translations allow us to establish equivalences between properties in each quadrant. For example, a semigroup is idempotent if and only if the left or right preorder is reflexive i.e. $x \preccurlyeq_L x \iff x = x \oplus x$.

We now define some examples of orders that will be used by our extended metalanguage RAML$_2$. These are in addition to the translations defined above.

Set size Let S be a set. Define the *set size* preorder as $(\mathcal{P}(S), \preccurlyeq)$, where for $X, Y \in T$, $X \preccurlyeq Y \iff |X| \leq |Y|$.

List length Define the *list length* preorder similarly to the set size preorder. Let S be a set, and let $T \in \{\mathcal{L}(S), \mathcal{S}(S)\}$. Define the preorder as $(T, \preccurlyeq)$, where for $X, Y \in T$, $X \preccurlyeq Y \iff |X| \leq |Y|$.

Trivial For a set S, define the *trivial* order over S as $(S, \sim)$, where for $s_1, s_2 \in S$, $s_1 \sim s_2$. i.e. all pairs of elements are related by this order.

Discrete For a set S, define the *discrete* order over S as $(S, =)$. That is, pairs of elements are only related if they are equal.

Top/bottom Let $S = (S, \preccurlyeq)$ be an order. Define the *top* order as $\top(S) = (1 + S, \preccurlyeq_\top)$, where for $x, y \in 1 + S$,

$$x \preccurlyeq_\top y \iff y = \text{inl}(1) \vee (x = \text{inr}(s_1) \wedge y = \text{inr}(s_2) \wedge s_1 \preccurlyeq s_2)$$

The *bottom* order is defined similarly, but with inl(1) now a minimal element.

Direct product Let $S = (S, \preccurlyeq_S)$ and $T = (T, \preccurlyeq_T)$ be preorders. Define the *direct product* preorder as $S \times T = (S \times T, \preccurlyeq_{S\times T})$, where for $(s_1, t_1), (s_2, t_2) \in S \times T$ we have that $(s_1, t_1) \preccurlyeq_{S\times T} (s_2, t_2) \iff s_1 \preccurlyeq_S s_2 \wedge t_1 \preccurlyeq_T t_2$. This can be extended to the n-ary case in a straightforward manner.

Lexicographic product Let $S = (S, \preccurlyeq_S)$ and $T = (T, \preccurlyeq_T)$ be preorders. Define the *lexicographic product* preorder as $S \vec{\times} T = (S \times T, \preccurlyeq_{S\vec{\times}T})$, where for $(s_1, t_1), (s_2, t_2) \in S \times T$ we have that $(s_1, t_1) \preccurlyeq_{S\times T} (s_2, t_2) \iff s_1 \prec_S s_2 \vee (s_1 = s_2 \wedge t_1 \preccurlyeq_T t_2.)$ Again, it is straightforward to extend this to the n-ary case.

Disjoint union Let $S = (S, \preccurlyeq_S)$ and $T = (T, \preccurlyeq_T)$ be preorders. Define the *disjoint union* preorder as $S + T = (S + T, \preccurlyeq_{S+T})$, where for $x, y \in S + T$,

$$x \preccurlyeq_{S+T} y = \begin{cases} s_1 \preccurlyeq_S s_2 & x = \text{inl}(s_1) \wedge y = \text{inl}(s_2) \\ t_1 \preccurlyeq_T t_2 & x = \text{inr}(t_1) \wedge y = \text{inr}(t_2) \end{cases}$$

Once more, it is straightforward to extend this to the n-ary case.

7.2.2 Transforms

We now discuss transforms. A transform (S, F) comprises a set of functions F, where each function $f \in F$ is of type $S \to S$. We are concerned with a restricted form of transform where the set of functions is indexed by some *label* set L. In this case, a transform has the form $(S, L, \triangleright)$ where $\triangleright\colon L \to S \to S$. We define a range of properties of transforms in Figure 7.7.

Our approach of specifying functions using labels, instead of directly as higher-order values, has two benefits. Firstly, policy checking becomes tractable; in general, function equality is uncomputable, and therefore it would be undecidable as to whether an arbitrary function f were indeed a member of F. Secondly, it allows policy to be more easily configured; it is far easier to write a first-order value $l \in L$ in a router configuration rather than a higher-order value. Policy configuration using labels is explored in more detail in [114].

Semigroups can be converted to transforms using a *Cayley* transform which 'partially applies' the semigroup operator. Suppose that we have a semigroup $(S, \oplus)$. We can

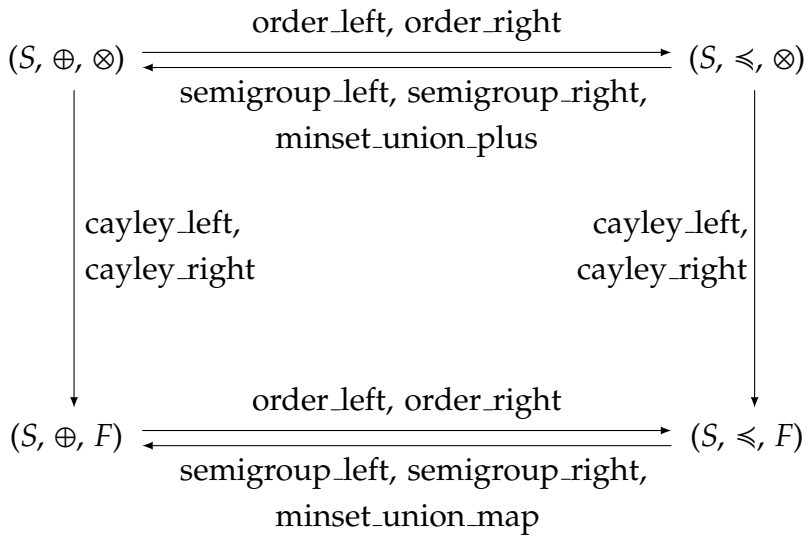


Figure 7.6: Translations between algebraic structures

Property	Definition
IDEM	$f(f(s)) = f(s)$
COMM	$f(g(s)) = g(f(s))$
INJ	$f(a) = f(b) \Rightarrow a = b$
CLOSE	$\exists h. f(g(s)) = h(s)$
CONST	$\exists c. f(s) = c$

Figure 7.7: Properties of transforms (S, F). All free variables are universally quantified.

convert it into a transform using the *left* Cayley transform as

$$\text{cayley\_left}(S, \oplus) = (S, S, \{\lambda x.s \oplus x \mid s \in S\}).$$

Similarly, the result of applying the *right* Cayley transform is

$$\text{cayley\_right}(S, \oplus) = (S, S, \{\lambda x.x \oplus s \mid s \in S\}).$$

We use these translations to convert from algebraic to functional policy application.

Transforms can also be converted to semigroups by using function composition over the function set. For example, consider the transform (S, F). This can be translated to a semigroup as $(F, \circ)$ where $\circ$ is the standard function composition operator i.e. for $f, g \in F$ and $s \in S$, $(f \circ g)(s) = f(g(s))$. We describe this translation for completeness, although we do not support it in the metalanguage due to the aforementioned problems with higher-order values.

We now define some examples of transforms that will be used by our extended metalanguage RAML$_2$. These are in addition to the translations defined above.

List consing Define the *listing consing* transform as $(S, \mathcal{L}(S), \text{cons})$, where S is a set and $\text{cons} \in S \to \mathcal{L}(S) \to \mathcal{L}(S)$ corresponds to the usual 'consing' operation, whereby an element is added to the head of a list.

Constant Define the *constant* transform as (S, S, const), where $\text{const} \in S \to S \to S$ is defined as $(\mathit{const}(s_1))(s_2) = s_1$ for $s_1, s_2 \in S$. That is, the partially-applied function $\text{const}(s)$ always returns $s \in S$ when applied to an additional argument.

Identity Define the *identity* transform as (S, S, id), where $\text{id} \in S \to S \to S$ is defined as $(\mathit{id}(s_1))(s_2) = s_2$ for $s_1, s_2 \in S$. The partially-applied function $\text{id}(s)$ performs as the identity function when applied to an additional argument.

Direct product Let $S = (S_1, S_2, \rhd_S)$ and $T = (T_1, T_2, \rhd_T)$ be transforms. Define the *direct product* transform as $S \times T = (S_1 \times T_1, S_2 \times T_2, \rhd_{S\times T})$, where for $(s_1, t_1) \in S_1 \times T_1$ and $(s_2, t_2) \in S_2 \times T_2$, we have $(s_1, t_1) \rhd_{S\times T} (s_2, t_2) = (s_1 \rhd_S s_2, t_1 \rhd_T t_2)$. This operation can be extended to the n-ary case in a straight-forward manner.

Disjoint union Let $S = (S_1, S_2, \rhd_S)$ and $T = (T_1, T_2, \rhd_T)$ be transforms. Define the *disjoint union* transform as $S + T = (S_1 + T_1, 1 + (S_2 + T_2), \rhd_{S+T})$, where for $(s_1, t_1) \in S_1 \times T_1$ and $(s_2, t_2) \in S_2 \times T_2$, we have

$$\begin{array}{rcll}
\text{inl}(s_1) & \rhd_{S+T} & \text{inr}(\text{inl}(s_2)) & = & \text{inr}(\text{inl}(s_1 \rhd_S s_2)) \\
\text{inr}(t_1) & \rhd_{S+T} & \text{inr}(\text{inr}(t_2)) & = & \text{inr}(\text{inl}(t_1 \rhd_T t_2)) \\
\text{inr}(_) & \rhd_{S+T} & \text{inr}(\text{inl}(_)) & = & \text{inl}(1) \\
\text{inl}(_) & \rhd_{S+T} & \text{inr}(\text{inr}(_)) & = & \text{inl}(1) \\
_ & \rhd_{S+T} & \text{inl}(1) & = & \text{inl}(1)
\end{array}$$

Again, this operation can be extended to the n-ary case in a straight-forward manner.

7.2.3 Minimal sets

We now describe a translation from preorders to semigroups when the preorder is not total. This was the case in the example in Section 7.1.1. The difficulty arises because there may be pairs of elements that are equivalent or incomparable, and in such cases there is no 'best' element. Our approach, described below, is to use *sets* of minimal values.

Suppose that we have a non-total preorder $(S, \preccurlyeq)$. Define *minimisation* on $X \in \mathcal{P}(S)$ as

$$\min_{\preccurlyeq}(X) = \{s_1 \in X \mid \forall s_2 \in X.\, s_2 \not\prec s_1\}$$

Using this operator, define the set of minimal sets from $\mathcal{P}(S)$ as

$$\mathcal{P}_{\preccurlyeq}(S) = \{X \in \mathcal{P}(S) \mid \min_{\preccurlyeq}(X) = X\}.$$

We now describe three operations that can be performed upon minimal sets. Firstly, for $X_1, X_2 \in \mathcal{P}_{\preccurlyeq}(S)$, define the *minimal set union* operation as

$$X_1 \cup_{\preccurlyeq} X_2 = \min_{\preccurlyeq}(X_1 \cup X_2).$$

This operation simply forms the union of the two argument sets and then removes any non-minimal elements.

Next, suppose that we have a transform (S, F). For $f \in F$ and $X \in \mathcal{P}_{\preccurlyeq}(S)$, define the *minimal set map* operation as

$$\text{map}_{\preccurlyeq}(f)(X) = \text{min}_{\preccurlyeq}(\{f(s) \mid s \in X\}).$$

This operation applies the function f to each element $x \in X$, and then removes non-minimal elements. In combination, the minimal set union and set map operations allow translation of a non-total order transform to a semigroup transform as

$$\text{minset\_union\_map}(S, \preccurlyeq, F) = (\mathcal{P}_{\preccurlyeq}(S), \cup_{\preccurlyeq}, \{\text{map}_{\preccurlyeq}(f) \mid f \in F\}).$$

Finally, suppose that we have a semigroup $(S, \oplus)$. For $X_1, X_2 \in \mathcal{P}_{\preccurlyeq}(S)$, define the *minimal set plus* binary operator as

$$X_1 \oplus_{\preccurlyeq} X_2 = \text{min}_{\preccurlyeq}(\{s_1 \oplus s_2 \mid s_1 \in X_1 \wedge s_2 \in X_2\})$$

That is, each pair of elements from X_1 and X_2 are combined using the $\oplus$ operator, and then any non-minimal values are removed from the resulting set. In combination, the minimal set union and set plus operations allow translation of a non-total order semigroup $(S, \preccurlyeq, \oplus)$ to a semigroup transform as

$$\text{minset\_union\_plus}(S, \preccurlyeq, \oplus) = (\mathcal{P}_{\preccurlyeq}(S), \cup_{\preccurlyeq}, \oplus_{\preccurlyeq}).$$

An extension to minimal sets is k-best sets [115]. Given a *total*-order $(S, \preccurlyeq)$, we define sets of at most k elements. Minimisation now consists of removing maximal elements one at a time until only k elements remain. This construction allows us to represent 'second-best' metrics etc. We can use k-best sets to translate from ordered summarisation to algebraic summarisation, in a similar manner to minimal sets. We return to this construction in Chapter 8.

7.3 Metalanguage

In this section we describe RAML$_2$, the extended metalanguage. The metalanguage contains constructors for all four kinds of algebras. A complete RAML$_2$ routing language specification comprises a sequence of `let`-bound sub-languages, as shown in the examples of Section 7.1. This facility allows complex routing languages to be decomposed into their constituent components. We give the syntax of RAML$_2$ in Figures 7.8, and 7.9.

Recall that RAML$_2$ is translated into the intermediate language IRL$_2$. This contains representations of order semigroups, semigroup transforms and order transforms. Whilst

we do not further elaborate upon the details, IRL$_2$ is a straight-forward, if not lengthy, extension of IRL$_1$. This language can then be given a semantics by mapping it into the mathematical domain of 7.2.

Compilation of RAML$_2$ is again a straight-forward extension of the bisemigroup case. We embed orders and transforms as functors within C++. This is similar to the manner in which semigroups are represented. Given a pair of elements (x, y), a functor that represents an order returns whether x is *less than*, *equivalent*, *greater than* or *incomparable* to y. This result type is encoded as an enumeration. Functors that represent transforms accept an addition label parameter.

The actual version of RAML that exists in the metarouting system is essentially the same as the RAML$_2$ metalanguage, with a few small additions. Firstly, the language incorporates more bounded types, such as bounded lists and strings. These types are essential for controlling the resource usage in deployed routing protocols; we might wish to control the maximum length of an AS path, for example. Secondly, we have added 'syntactic sugar', such as facilities for allowing fields to be renamed and reordered in records. Whilst this does not affect the underlying semantics of the language, we believe that it aids usability. Finally, as noted in the regions example (§ 7.1.2), in some cases we allow record fields to be omitted if they are unused.

ty	::=		(type)
	|	...	
	|	`minset(`*ty*`,`*ord*`)`	(minimal set)
	|	`kbest(`*n*`,`*ty*`,`*ord*`)`	(*k*-best set)
ord	::=		(order)
	|	`list_len(`*ty*`)`	(list length)
	|	`list_simp_len(`*ty*`)`	(simple list length)
	|	`set_size(`*ty*`)`	(set size)
	|	`flip(`*ord*`)`	(flipped order)
	|	`triv(`*ty*`)`	(trivial order)
	|	`disc(`*ty*`)`	(discrete order)
	|	`add_top(`*i*`,`*ord*`)`	(add top)
	|	`add_bot(`*i*`,`*ord*`)`	(add bottom)
	|	`dir_prod(`i_1`:`ord_1`,...,`i_n`:`ord_n`)`	(direct product)
	|	`lex_prod(`i_1`:`ord_1`,...,`i_n`:`ord_n`)`	(lexicographic product)
	|	`disj_union(`i_1`:`ord_1`,...,`i_n`:`ord_n`)`	(disjoint union)
	|	`sg_left(`*sg*`)`	(left semigroup translation)
	|	`sg_right(`*sg*`)`	(right semigroup translation)
bsg	::=		(bisemigroup)
	|	...	...
	|	`sg_left(`*osg*`)`	(left semigroup translation)
	|	`sg_right(`*osg*`)`	(right semigroup translation)
	|	`minset_union_plus(`*osg*`)`	(minimal set union / plus translation)
	|	`kbest_union_plus(`*n*`,`*osg*`)`	(*k*-best set union / plus translation)
osg	::=		(order semigroup)
	|	`set_sub_union(`*ty*`)`	(set subset / union)
	|	`list_lte_app(`*ty*`)`	(list length / append)
	|	`list_simp_lte_app(`*i*`,`*ty*`)`	(simple list length / append)
	|	`add_top_omega_times(`*i*`,`*osg*`)`	(add top / omega times)
	|	`add_bot_alpha_times(`*i*`,`*osg*`)`	(add bottom / alpha times)
	|	`dual(`*osg*`)`	(dual order)
	|	`dir_prod(`i_1`:`osg_1`,...,`i_n`:`osg_n`)`	(direct product)
	|	`lex_prod(`i_1`:`osg_1`,...,`i_n`:`osg_n`)`	(lexicographic product)
	|	`disj_union(`*i*`,`i_1`:`osg_1`,...,`i_n`:`osg_n`)`	(disjoint union)
	|	`order_left(`*bsg*`)`	(left order translation)
	|	`order_right(`*bsg*`)`	(right order translation)

Figure 7.8: Syntax of RAML$_2$ types, orders, bisemigroups and order semigroups

sgt	::=		(semigroup transform)
	|	`const(`*sgt*`)`	(constant)
	|	`id(`*sgt*`)`	(identity)
	|	`dir_prod(`i_1`:`sgt_1`,...,`i_n`:`sgt_n`)`	(direct product)
	|	`lex_prod(`i_1`:`sgt_1`,...,`i_n`:`sgt_n`)`	(lexicographic product)
	|	`disj_union(`i`,`i_1`:`sgt_1`,...,`i_n`:`sgt_n`)`	(disjoint union)
	|	`cayley_left(`*bsg*`)`	(left Cayley translation)
	|	`cayley_right(`*bsg*`)`	(right Cayley translation)
	|	`sg_left(`*ot*`)`	(left semigroup translation)
	|	`sg_right(`*ot*`)`	(right semigroup translation)
	|	`minset_union_map(`*ot*`)`	(minimal set union / map translation)
	|	`kbest_union_map(`n`,`*ot*`)`	(k-best set union / map translation)
ot	::=		(order transform)
	|	`const(`*ot*`)`	(constant)
	|	`id(`*ot*`)`	(identity)
	|	`dual(`*ot*`)`	(dual order)
	|	`list_lte_cons(`*ty*`)`	(list length / cons)
	|	`list_simp_lte_cons(`i`,`*ty*`)`	(simple list length / cons)
	|	`add_top_fix(`i`,`*ot*`)`	(add top / fixed-point)
	|	`add_bot_fix(`i`,`*ot*`)`	(add bottom / fixed-point)
	|	`dir_prod(`i_1`:`ot_1`,...,`i_n`:`ot_n`)`	(direct product)
	|	`lex_prod(`i_1`:`ot_1`,...,`i_n`:`ot_n`)`	(lexicographic product)
	|	`disj_union(`i`,`i_1`:`ot_1`,...,`i_n`:`ot_n`)`	(disjoint union)
	|	`cayley_left(`*osg*`)`	(left Cayley translation)
	|	`cayley_right(`*osg*`)`	(right Cayley translation)
	|	`order_left(`*sgt*`)`	(left order translation)
	|	`order_right(`*sgt*`)`	(right order translation)
b	::=		(binding)
	|	`let` i `: semigroup =` *sg*	(semigroup)
	|	`let` i `: order =` *ord*	(order)
	|	`let` i `: bisemigroup =` *bsg*	(bisemigroup)
	|	`let` i `: order_semigroup =` *osg*	(order semigroup)
	|	`let` i `: semigroup_transform =` *sgt*	(semigroup transform)
	|	`let` i `: order_transform =` *ot*	(order transform)
s	::=	$\{b\}^+$	(routing language specification)

Figure 7.9: Syntax of RAML$_2$ semigroup transforms, order transforms, bindings and routing language specifications

Chapter 8

Performance

In this section we present a number of optimisation techniques for improving the runtime performance of compiled routing languages. These optimisation techniques may be specified when invoking the metarouting compiler. We commence with a discussion of the different optimisation techniques (§ 8.1), and the test methodology (§ 8.2). We then present the results of the performance experiments (§ 8.3). Finally, we relate the results to the optimisation techniques and discuss the relevance of the results to online routing (§ 8.4).

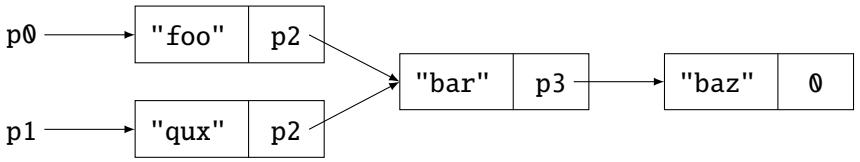
8.1 Optimisations

8.1.1 Overview of sharing

Our first optimisation technique is known as *sharing* and involves storing multiple equivalent values just once within the runtime. We are able to use this optimisation technique because state is *immutable* (i.e. cannot be changed) within compiled routing languages. Sharing can be applied to any routing language, although with varying degrees of success according to the underlying data-structures and runtime values.

There are two main benefits to sharing. Firstly, providing that there are indeed a sufficient number of duplicate values, it can reduce memory usage. For example, [116] has an example of a lambda evaluator where memory usage decreases by two orders of magnitude. Secondly, sharing opens up a number of avenues for increasing execution speed by identifying values according to their location in memory.

Sharing was first widely used to increase the efficiency of implementations of the LISP functional programming language. The technique was known as *hash-consing* [117], due to the fact that `cons` is the only operation in LISP that allocates values on the heap, and the sharing is typically implemented using hash tables. More recently, [116] has demonstrated how the technique can also be implemented as a library for the OCaml



(a) List representation

Key	Value
("foo", p2)	p0
("qux", p2)	p1
("bar", p3)	p2
("baz", 0)	p3

(b) Hash table

Figure 8.1: Example of hash-consed lists

language. Sharing can also be applied to object oriented programs in the form of the *fly-weight* pattern [118]. Common object state is factored into external structures that are typically reference-counted or garbage-collected. Hence multiple objects can reference the same common state, often with the storage overhead of only a single pointer per shared value.

We note that sharing is already used in existing routing implementations to reduce memory usage. For example, the Quagga BGP daemon stores all AS paths within a hash-table. Whenever an AS path is received from a peer, the hash-table is first checked to see whether that AS path is already present within the system. If it is, then a reference to the existing AS path is returned (avoiding duplication), otherwise the new AS path is stored in the hash-table. Therefore this section demonstrates how to systematically apply an optimisation technique that is currently implemented 'by hand' in existing router platforms.

One important property of systems implementing sharing is *maximal* sharing. A system implementing this form of sharing guarantees that *all* pairs of semantically equal values are shared [116]. This means that semantic equality coincides with physical equality and therefore semantic equality can be cheaply checked with a single pointer equality test; values are semantically equal if and only if they reside at identical locations in memory.

We exploit sharing in several ways. Firstly, in Section 8.1.2, we describe our implementation of sharing for lists. This uses a simple hash-consing mechanism. Next, in Section 8.1.3, we show how to efficiently implement shared sets using a data structure known as a *Patricia tree*. Our design is dependent upon maximal sharing of element values for efficient operation. Finally, in Section 8.1.4, we show how to *memoise* operator implementations so that they cache result values. Again, for efficient operation, our design requires that operator arguments are maximally shared.

8.1.2 Hash-consed lists

In this section we describe how we using hash-consing to share lists. We represent lists as chains of heap-allocated pairs. Each pair comprises a list element value and a

pointer to the tail of the list (or the null-pointer, `0`, if there is none). We call these pairs 'cons cells' after the analogous structures found in LISP. Consider the example of hash-consed lists shown in Figure 8.1(a). Here there are two lists, `["foo","bar","baz"]` and `["qux","bar","baz"]`, which are accessible via pointers `p0` and `p1` respectively.

A benefit of hash-consing is that it allows sharing of sub-structure; even though the two lists in Figure 8.1(a) are non-identical, their common tails are still shared. This permits a greater degree of sharing over schemes such as that found in Quagga, whereby pairs of lists are shared only if they are identical. In the latter approach, minor variations between lists cause a loss of sharing.

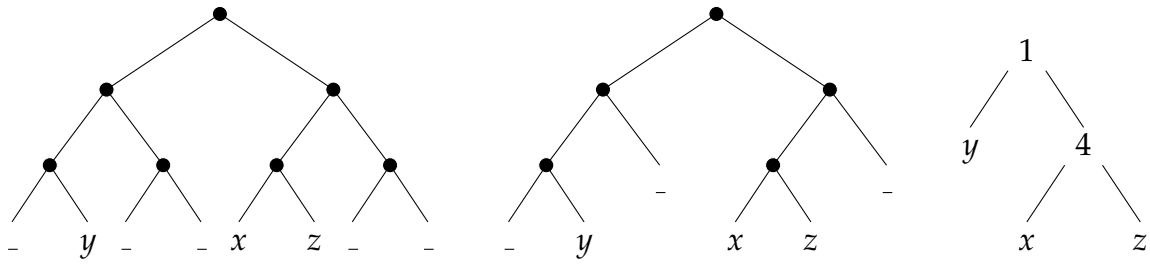
We implement sharing using a hash-table. A hash table is an efficient map structure that uses a hash function to permit $O(1)$ lookups and insertions (with some caveats). Keys comprise pairs of a list element value and a pointer, whilst values are pointers to the corresponding heap-allocated cell. When constructing lists, we start with the last element and work backwards. Given a value and a tail pointer, the hash-table is checked to see whether a corresponding cons cell is already allocated. If so, a pointer to the existing cons cell is returned. Otherwise, a new cons cell is allocated, and an entry is inserted into the hash-table. Figure 8.1(b) illustrates the hash-table associated with our example lists.

In order to prevent memory leaks, it is necessary to detect when cons cells are no longer accessible and remove them from the heap. For this, we use the standard technique of *reference counting*. Each cons cell is associated with an integer reference count indicating the number of 'incoming' pointers. The reference count can either be stored in the hash-table or else in the cons cells themselves. Whenever the reference count drops to zero, the cons cell can be removed from the heap. Note that in languages with automatic memory management, such as OCaml, the task of freeing unreferenced cons cells may be delegated to the garbage collector.

A possible improvement is to leave such cells allocated and only reclaim the heap space when the amount of available free memory on the system reaches a specified minimum value ('weak' pointers can be used for this purpose). This strategy can reduce the likelihood of cons cells being deleted and then subsequently recreated. We do not further explore this avenue.

8.1.3 Patricia trees

In this section we describe how we efficiently represent sets of values using a data structure known as a Patricia tree [119]. A Patricia tree is an example of a more general data structure known as a trie. Tries are tree-like structures that are used for the efficient implementation of maps. Keys are grouped according to their prefixes, leading to good spatial locality for lookups and insertions. Furthermore, empty subtrees can



(a) Binary trie (b) Binary trie, no empty sub-trees (c) Patricia tree

Figure 8.2: Comparison between binary tries and Patricia trees. Each structure represents the 3-bit map $\{1 \to x, 4 \to y, 5 \to z\}$. Keys are processed in their binary representation, least-significant bit first e.g. 1, 4 and 5 are processed as the strings 100, 001 and 101 respectively. Example taken from [120].

be eliminated in order to increase memory efficiency. It is for these reasons that tries are frequently used within router implementations when storing IP addresses.

Figure 8.2 illustrates binary tries and Patricia trees. Figure 8.2(a) shows a standard binary trie. Map keys are processed bit-wise, with the bit value determining whether to descend into the left or right sub-tree. Figure 8.2(b) shows the effect of eliminating empty sub-trees. A Patricia tree additionally eliminates redundant bit tests, with each node now containing a value specifying the bit position to test. The Patricia tree in Figure 8.2(c) represents bit position n as the value 2^n. Hence, when finding a value, we first test the zeroth bit (node value $2^0 = 1$) of the key. If it is zero, then the only value in the left sub-tree is y, and hence we can immediately test the key against that stored in the leaf node for y (leaf keys omitted). Otherwise, we test the second bit-position (node value $2^2 = 4$). Depending upon the result, we can immediately test the key against that stored in either the left or the right leaf nodes.

In order to represent sets of values using Patricia trees, it is first necessary to associate each value with an integer key. This allows values to be inserted into Patricia trees. Note that the association between each key and value must be unique in order to prevent a single value from being stored multiple times within the same Patricia tree. For maximally-shared values, we obtain such an association by using the memory addresses of the values; by the maximal sharing property, each value is guaranteed to be represented just once, and hence have a unique memory address. For values that are not maximally-shared we can instead use hashes, although they may be more expensive to compute and may also reduce the locality of storage within the Patricia trees.

In common with [116], we implement maximal-sharing of Patricia trees themselves by sharing sub-trees within Patricia trees. This requires that each node additionally contains a reference count, although in most cases the additional cost appears to be negligible. Maximally-shared Patricia trees can then be stored within other Patricia

```
int times_two(int x)
{
  return x * 2;
}
```

(a) Double function

Key	Value
3	6
19	38
2	4

(b) Hash table

Figure 8.3: Example of memoisation

trees. In this way, we can efficiently represent sets of sets etc. (see below for an example).

Micro-benchmarks from [120] suggest that Patricia trees support particularly fast merging when compared with binary search trees, splay trees and red-black trees. When using tries to represent sets, this leads to efficient set union operations. This is an improvement upon the GNU implementation of the C++ STL set, which internally uses red-black trees. Patricia trees are also competitive with other tree datatypes for insertion and lookup times, especially when keys are processed most-significant bit first (this tends to increase locality of storage within the tree).

8.1.4 Memoisation

In this section we describe an optimisation strategy called *memoisation*. In contrast to the list and set data-structure optimisations presented in Sections 8.1.2 and 8.1.3, this approach is applied to functions. This optimisation caches argument/result pairs of functions to avoid re-evaluating the functions at identical arguments. Typically the cache is implemented using a hash table.

Consider the `times_two` function in Figure 8.3(a), which returns its integer argument doubled. When evaluating `times_two` at a given value x, the associated hash-table is first consulted to check whether the function has previously been evaluated at this value. If so, then the cached result value is returned. Otherwise, the function is evaluated and the result value is stored in the hash table, before being returned to the function caller. The cache in Figure 8.3(b) indicates that `times_two` has already been evaluated at `2`, `3` and `19`. Re-evaluating the function at any of these values would immediately return the cached result values without invoking `times_two`.

Memoisation can be implemented particularly efficiently for maximally-shared values. Here, keys and values within the hash-table comprise pointers. Checking equality of two keys is reduced to a single pointer equality check. Also, note that greater benefits are obtained when memoising particularly 'expensive' functions. For 'cheap' functions, the cost associated with maintaining the cache table may outweigh the benefits.

A disadvantage of memoisation is that it introduces a speak leak: all key/value pairs remain cached until the program terminates. One alternative is to remove key/value

pairs from the hash-table as soon as a key or value is deleted from the system. However, this often leads to very little caching. Therefore the preferred solution is to instead use weak pointers, as outlined for hash-consed lists. We do not further discuss this feature.

8.1.5 Minimisation

In this section we discuss two algebraic optimisations for algebras that use minimal sets. In comparison to the optimisations that are based upon sharing, each of the optimisations for minimal sets is dependent upon properties that are automatically inferred by the compiler.

Avoiding minimisation

Our first optimisation for minimal sets allows the minimisation operation to be elided. It exploits the fact that some operations upon minimal sets produce sets which are themselves already minimal. Concretely, suppose that we have an order transform $S = (S, \preccurlyeq, F)$. Convert it to a semigroup transform as $(T, \oplus, G) = \text{minset\_union\_map}(S)$. We can avoid minimisation after mapping a function over a set if for all $f \in F$ and $x, y \in S$, we have the property

$$(x \sim y) \vee (x \sharp y) \Rightarrow (f(x) \sim f(y)) \vee (f(x) \sharp f(y)).$$

That is, pairs of elements that are incomparable or equivalent remain incomparable or equivalent after function application. Depending upon the computational cost of the minimisation operation, this can lead to a significant reduction in the overall execution time.

One example of a preorder transform that has this property is `list_lte_cons(`*ty*`)` i.e. lists of elements of type *ty*, with shorter lists preferred and function application corresponding to the cons operation. Pairs of lists that are of equal length remain of equal length after applying a cons operation. Therefore, upon converting to a semigroup transform as `minset_union_map(list_lte_cons(`*ty*`))`, it is safe to elide the minimisation operation after mapping a cons operator over a (minimal) set.

Lazy minimisation

Our next optimisation allows us to 'lazily' minimise sets. That is, it is safe to omit minimisation operations to obtain non-minimal sets, providing that minimisation eventually occurs. Note that this may lead to a *decrease* in performance due to non-minimal elements increasing the average set size.

This optimisation again assumes a preorder transform $S = (S, \preccurlyeq, F)$. Convert it to a semigroup transform as $(T, \oplus, G)$ = minset_union_map(S). We can temporarily avoid minimisation after mapping a function over a set if for all $f \in F$ and $x, y \in S$, we have the *monotonicity* property

$$x \preccurlyeq y \Rightarrow f(x) \preccurlyeq f(y).$$

Monotonicity implies that the relative ordering of elements remains unchanged after function application. Therefore a non-minimal element will always remain non-minimal, even after function application, and hence its removal from a set may be safely delayed.

The preorder transform `list_lte_cons(`ty`)` has the monotonicity property; for lists X and Y, we have that $|X| \leq |Y| \Rightarrow |\text{cons}(z)(X)| \leq |\text{cons}(z)(Y)|$. Therefore, upon converting to the semigroup transform `minset_union_map(list_lte_cons(`ty`))`, it is safe to temporarily elide the minimisation operation (although we have already shown in the previous section that minimisation can *always* be elided in this case).

8.2 Methodology

In order to evaluate the effects of different optimisation techniques, it is necessary to accurately measure the runtime performance of compiled routing languages. Our approach is to combine the compiled routing languages with a standard offline algorithm in order to create an executable program. We do *not* use an online algorithm because our intention is to measure the performance of the routing language only, and offline algorithms permit a more controlled test environment (for example, we do not have to control for issues such as network latency or packet buffer sizes).

8.2.1 Routing languages

We evaluate optimisations using three different routing languages. The first two languages are intended to be toy examples of languages for Internet routing protocols. It is plausible that these languages might be elaborated to produce fully-fledged routing protocols, and therefore the optimisation of these languages is of particular interest. The third language is drawn from an operational research context. This example is useful for assessing the generality of our optimisation techniques, and also demonstrates that metarouting may have applications beyond Internet routing.

Set of minimal paths

The first routing language is based upon the minimal paths example from from Section 7.1.1. The routing language is defined as

```
let paths : order_semigroup = list_simp_lte_app(NOTSIMP, string)
let min_paths : bisemigroup = minset_union_plus(paths)
```

That is, elements of the language comprise sets of duplicate-free, minimal-length paths. We evaluate this routing language over graphs in which each arc (i, j) is labelled as `{["`i`-`j`"]}`.

K-best paths

Our second algebra, `kbest_paths`, builds upon `min_paths` in two ways. Firstly, we associate each path with a distance and bandwidth. These values are then compared lexicographically. This routing algebra is defined as

```
let paths : order_semigroup = list_simp_lte_app(NOTSIMP, string)
let dist  : order_semigroup = order_left(min_plus_bound(ERR, 0, 100))
let bw    : order_semigroup = order_left(max_min_bound(ERR, 0, 1000))
let dist_bw_paths : order_semigroup =
  lex_prod(
    dist : dist,
    bw   : bw,
    path : paths,
  )
```

In a practical setting, the addition of distance and bandwidth metric might provide a finer degree of control over selected routes. Secondly, we compute k-best sets of paths instead of minimal sets of paths:

```
let kbest_paths : bisemigroup = kbest_union_plus(5, dist_bw_paths)
```

In this particular instance, we constrain the maximum number of paths to be five. Hence values in this routing language may contain up to four sub-optimal paths in addition to the best path(s).

We evaluate this routing language over graphs in which each arc (i, j) is labelled as `{<dist = 1, bw = 1, path = ["`i`-`j`"]>}`.

Martelli's algebra

Our final algebra, `martelli`, is from [121, 122]. We again define this algebra in two steps. Firstly we define a preorder semigroup `sets` comprising sets of strings, ordered by set inclusion (subsets are preferred). Sets are combined with the union operation. This routing language is defined as

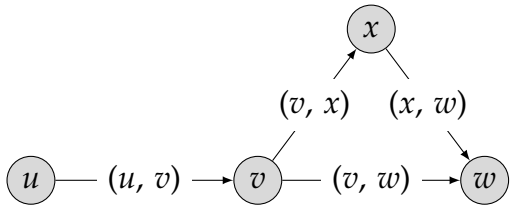


Figure 8.4: Example graph to illustrate minimal cut-sets

```
let sets : order_semigroup = set_sub_union(string)
```

We then translate `sets` to a bisemigroup using a minset construction, and then exchange the additive and multiplicative components:

```
let martelli : bisemigroup = swap(minset_union_plus(sets))
```

The resulting algebra comprises sets of sets, satisfying the condition that within each set, no set is a subset of any other. Multiplication corresponds to set union whilst addition is set plus, with each set element unioned.

The Martelli routing language is used to compute sets of minimal cut-sets. Suppose that we have a graph $G = (V, E)$. Then for $i, j \in V$, an $i - j$ cut-set is a set of edges $E' \subseteq E$ that *disconnects* i from j i.e. there is no path from i to j in $G' = (V, E \setminus E')$. A *minimal* $i - j$ cut-set is an $i - j$ cut-set E' for which there is no $i - j$ cut-set $E'' \subset E'$.

We illustrate minimal cut sets using the graph in Figure 8.4. The full set of minimal (u, w) cut-sets comprises

$$\{\{(u, v)\}, \{(v, x), (v, w)\}, \{(x, w), (v, w)\}\}.$$

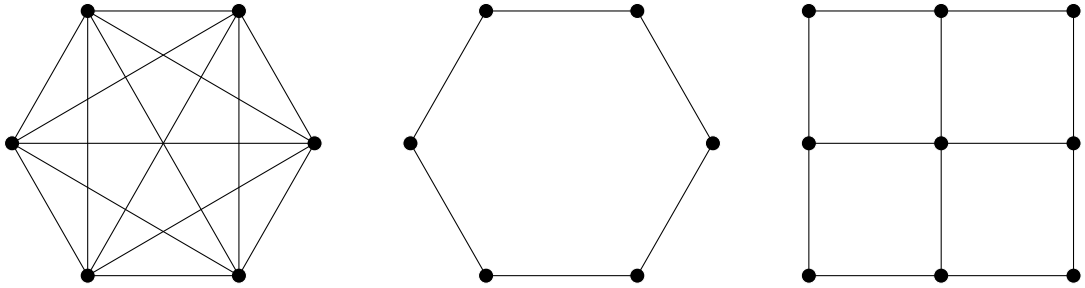
That is, in order to disconnect node u from node w, it is sufficient to either (i) remove arc (u, v), or (ii) arcs (u, v) and (v, w), or (iii) arcs (x, w) and (v, w). These cut-sets are minimal in the sense that no edges can be removed from any of them without reconnecting u to w.

We evaluate this routing language over graphs in which each arc (i, j) is labelled as `{{"`*i*`-`*j*`"}}`.

8.2.2 Optimisations

We compile each routing language with three different general optimisations:

std This is the unoptimised version against which we measure the relative performance of the rest of the optimisations.



(a) clique(6) (b) ring(6) (c) grid(3)

Figure 8.5: Examples of *clique*, *ring* and *grid* topologies

share This optimisation enables maximal sharing of values. For example, strings are shared using a hash-table, whilst lists are hash-consed and sets are represented as Patricia trees.

memo This optimisation additionally enables function memoisation over maximally shared datatypes. For example, both the list append and also set union operators are memoised.

For the `min_paths` routing language we additionally include the following algebraic optimisation:

no_min *In addition to enabling the 'memo' optimisations*, this optimisation eliminates unnecessary set minimisations using the techniques described in Section 8.1.5.

8.2.3 Graphs

In general, the runtime performance of offline algorithms depends upon the statistical properties of the graphs over which they are executed. For example, we expect graphs with greater numbers of nodes and/or greater proportions of nodes with high degrees to increase the time and memory requirements for the solution. We therefore control for these factors by running our tests over a fixed range of topologies. Note that we do not use random graph models, such as the Erdõs-Rényi [123] model because the statistical properties of the generated graphs varies widely. We are also not concerned with running our algorithms over realistic Internet topologies, because the generation of such topologies remains an active area of research in its own right.

The first kind of graph uses the *clique* topology, illustrated in Figure 8.5(a). Here, every node is connected to every other node (links are bidirectional in each of our topologies). We denote the clique topology with n nodes by clique(n). This topology illustrates the effects of large numbers of paths; there are $n!$ simple paths in a clique with n nodes. We note that whilst each pair of distinct nodes within the clique topology is directly

connected, the distribution of (simple) path lengths is heavily weighted towards longer paths i.e. paths that traverse a greater number of arcs.

The next kind of graph uses the *ring* topology. We show an example of a ring topology in Figure8.5(b). Here, every node is connected to exactly two other nodes to form a cycle. We denote the ring topology with n nodes by ring(n). There are exactly two simple paths between each pair of distinct nodes in the ring topology. Therefore this topology demonstrates the effect of small numbers of paths.

The final kind of graph uses the *grid* topology, which is shown in Figure 8.5(c). The nodes in a grid topology are connected to form a square array. Each internal node has four connections to its neighbours. We denote the $n \times n$ grid topology by grid(n). Compared to the clique topology, the grid topology has a greater proportion of paths of a shorter length. The grid topology also has a greater proportion of minimal length paths when compared to the ring topology. For grid(3), shown in Figure 8.5(c), there are six simple paths of minimal length between diagonally opposite edge nodes.

We test the scaling effects of different optimisations by altering the sizes of our chosen topologies. Some routing languages require relatively large topologies (e.g one hundred nodes) in order for us to observe meaningful time and memory characteristics. Larger topologies also allow us to amortise the costs of any overheads such as program startup costs. We programmatically generate each topology size to facilitate the testing.

For more 'interesting' datatypes such as sets, lists and products, we expect that the distribution of data used to label arcs may affect the time and memory characteristics of the offline algorithms. Therefore we control for this factor by using fixed arc labellings. We specify the particular arc labelling when describing each routing language.

8.2.4 Measurement

All tests are performed on a 32-bit GNU/Linux operating system running the Linux 2.6.26-2 SMP kernel. The generated code is compiled using GCC 4.3.2 with `-O2`, enabling all optimisations that do not increase executable size. The test hardware is a 1.83GHz Intel Core Duo with 1GB of RAM.

Timing data is obtained using the `getrusage()` system call at end of each test run. Memory usage is read using the `VmHWM` field from `/proc/self/status`. This tells us the virtual memory 'high water mark' (maximum) usage. The value is read both at the start and the end of each test run so that we can subtract the constant amount of memory used by the program code and static data.

Each test is repeated five times to obtain an average timing value. The memory usage remains constant for repeated tests.

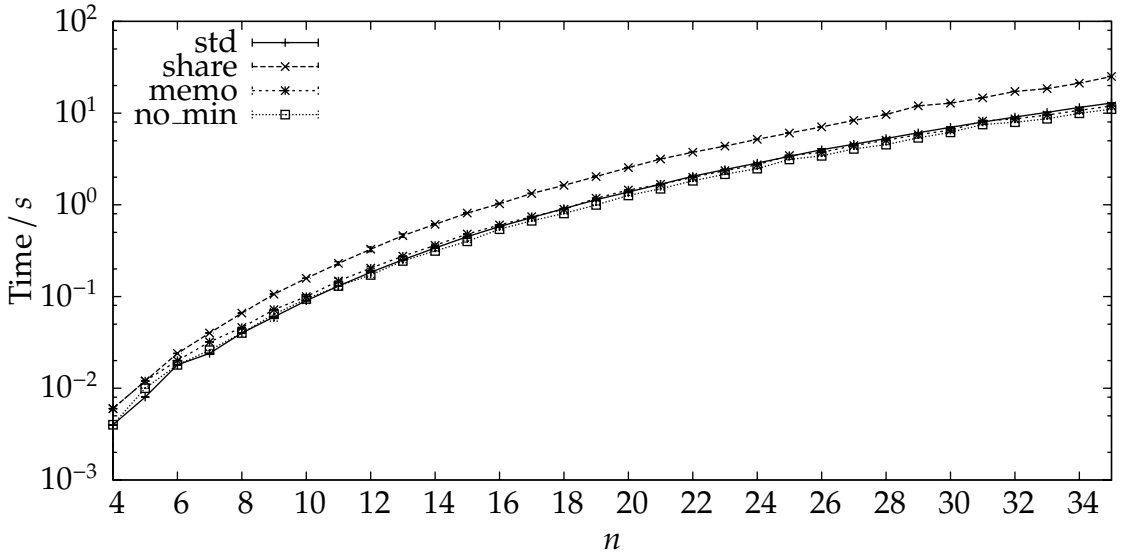
8.3 Results

8.3.1 Minimal-length paths

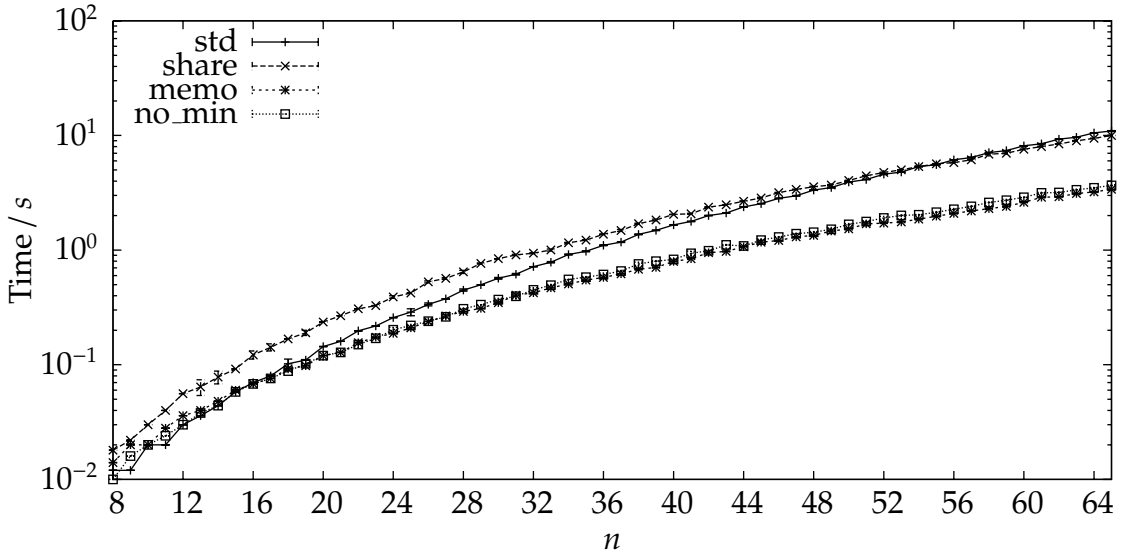
For the clique topology, there is a single minimal-length path between each pair of nodes, and therefore each minimal set comprises a one-element list. We first examine the time characteristics, shown in Figure 8.6(a). Here, the 'memo' and 'no_min' optimisations only give a marginal benefit, whilst the 'share' optimisation gives a decrease in performance. We hypothesise that the combination of the small set and list sizes mean that most operations consume relatively little time even in the 'std' version, and hence there is little scope for optimisation. Turning to the memory usage, shown in Figure 8.7(a), we see that memoisation has a large space penalty (recall that this optimisation effectively introduces a space leak). The 'no_min' optimisation reduces this overhead by causing fewer inputs to be cached.

The ring topology has longer minimal-length paths on average than the clique topology of the same degree. However, with the exception of diametrically opposite nodes, there remains only a single minimal-length path between each pair of nodes. We hypothesise that the longer average path lengths mean that the list append and length comparison operations tend to be more expensive, and hence there is greater scope for optimisation. Examining the time characteristics, shown in Figure 8.6(b), we see that memoisation indeed gives a substantial time saving. The memory characteristics, presented in Figure 8.7(b), show that sharing approximately halves the memory usage for a ring of size 65. These substantial savings can be attributed to the low path diversity causing a high degree of sharing.

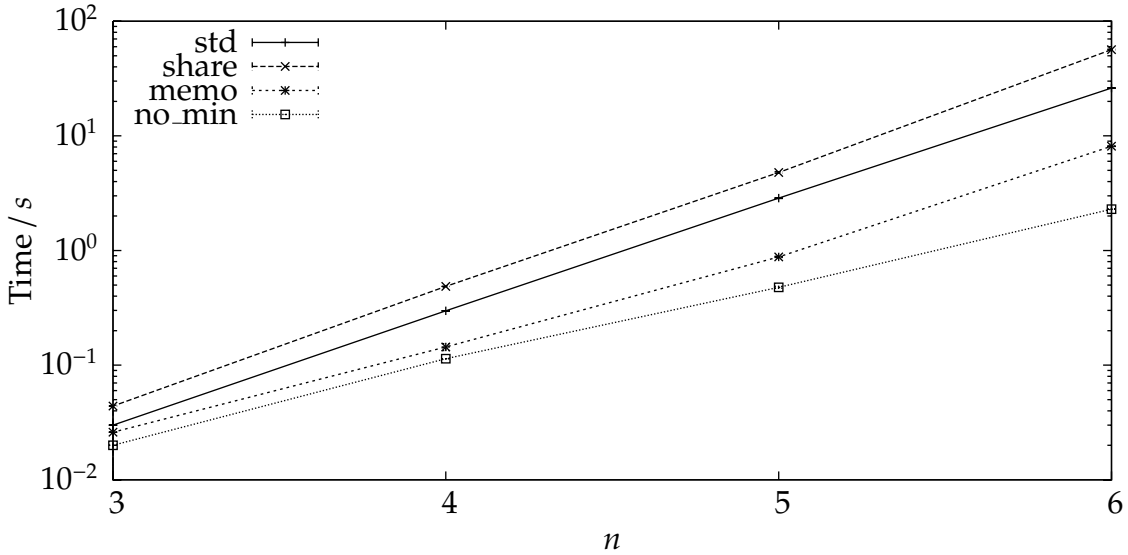
The grid topology has a greater number of minimal-length paths. The corresponding increase in average set size allows the `no_min` optimisation to give a substantial time saving, as shown in Figure 8.6(c). The increase in path diversity eliminates the space savings from sharing, as shown in Figure 8.7(c).



(a) Clique topology

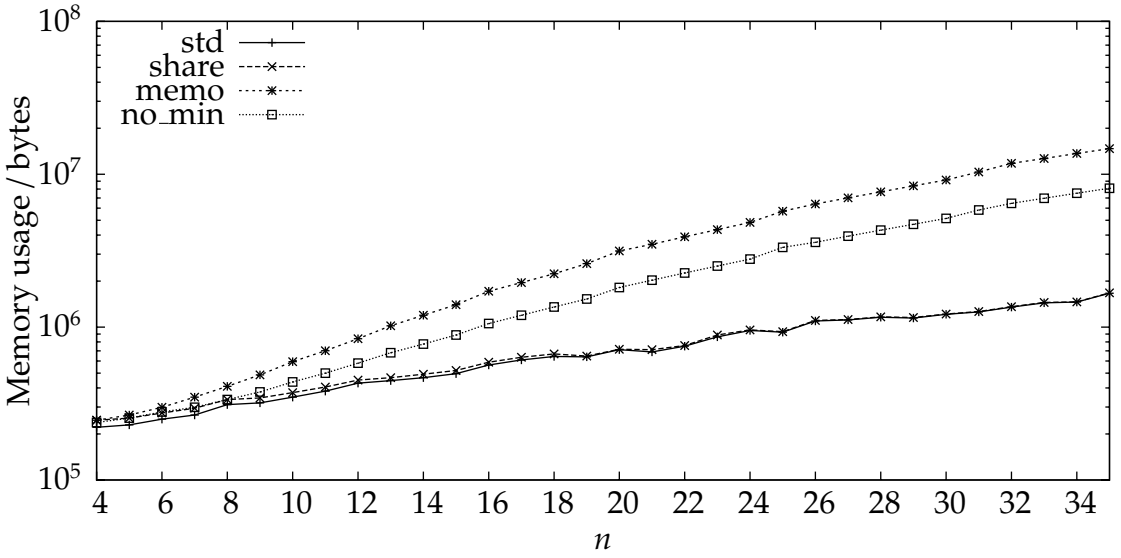


(b) Ring topology

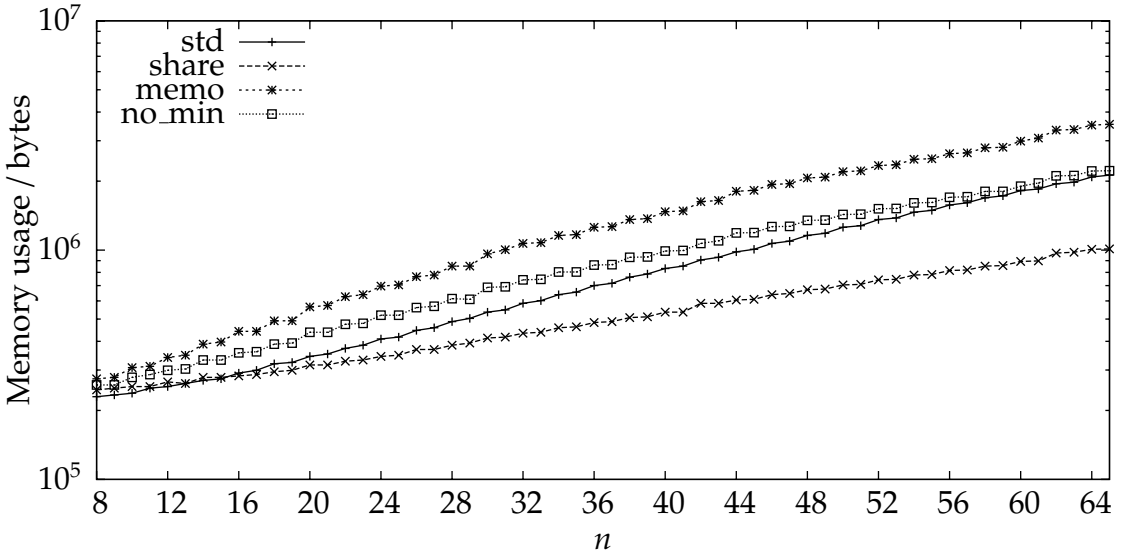


(c) Grid topology

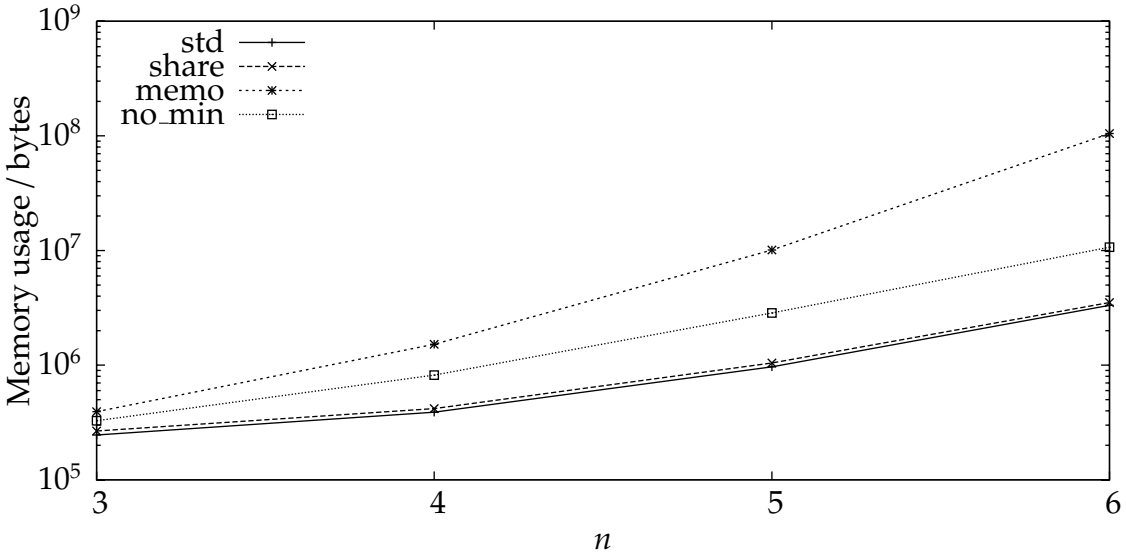
Figure 8.6: CPU usage for minimal-length paths routing language.



(a) Clique topology



(b) Ring topology



(c) Grid topology

Figure 8.7: Memory usage for minimal-length paths routing language.

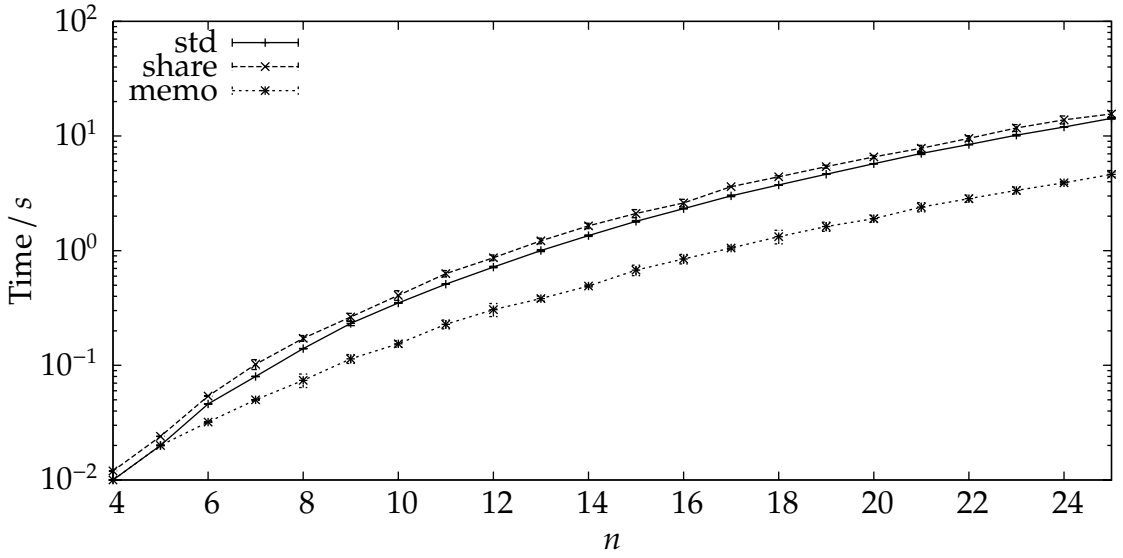
8.3.2 *K*-best paths

The results from the *k*-best paths routing language are broadly similar to those from the minimal-length paths routing language. The main differences can be explained by the fact that we also admit up to $k - 1$ sub-optimal paths into each set (in this instance, $k = 5$). Therefore the the ring and clique topologies see an appreciable increase in the average set size, giving greater scope for optimisation in these cases.

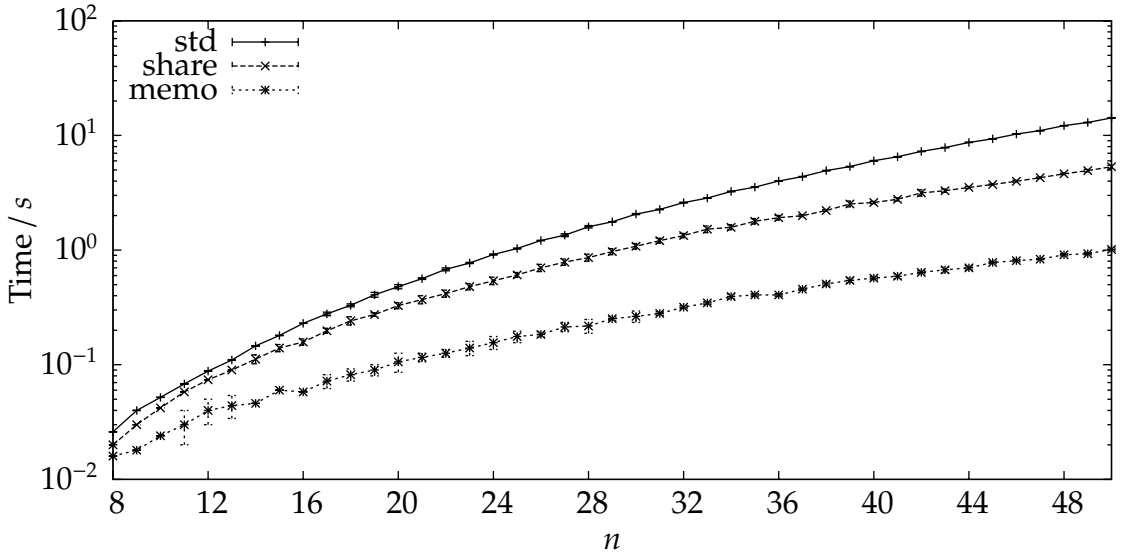
For the clique topology times, shown in Figure 8.8(a), we see that memoisation does indeed give an appreciable increase in performance in comparison to the minimal-lengths paths example. Turning to the memory performance, shown in Figure 8.9(a), the increase in memory usage for memoisation can be attributed both to the larger set sizes as well as a greater number of different sets.

We now examine the performance for the ring topology. A notable change from the minimal-length paths routing language is that sharing now gives an appreciable decrease in execution time. This is shown in Figure 8.8(b). We hypothesise that this is because there is greater scope for benefiting from sharing for list comparisons; comparisons on identical lists now comprise just a single pointer equality test, and there are a greater number of such tests due to the larger set sizes. Memory usage, shown in Figure 8.9(b), remains similar to that for the minimal-length paths routing language.

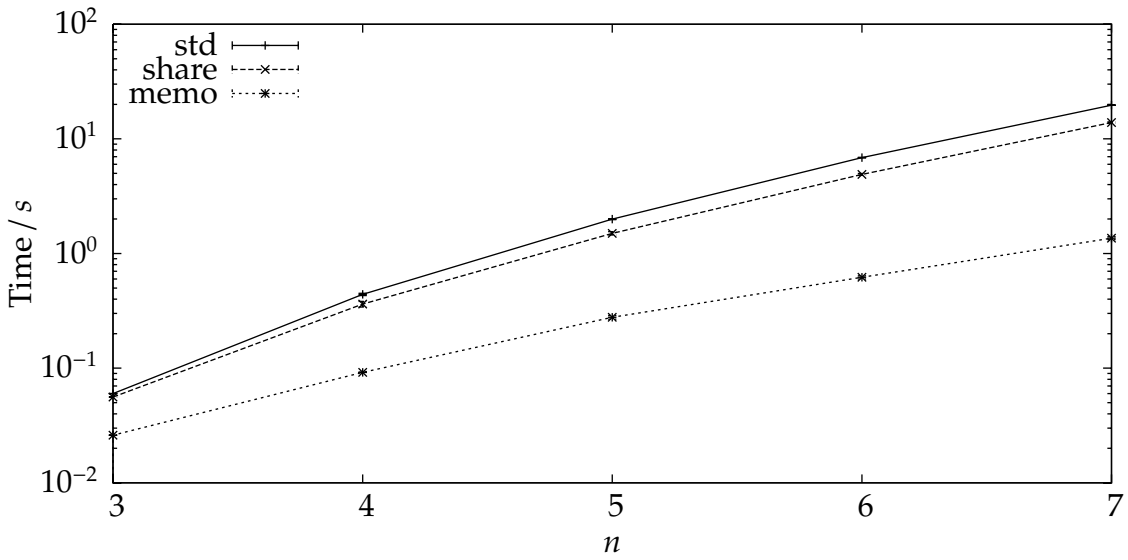
Finally, we examine the performance for the grid topology. Again, in comparison to the minimal-length paths routing language, sharing now gives a slight decrease in execution time. This is shown in Figure 8.8(c). Memory usage, shown in Figure 8.9(c). again remains similar to that for the minimal-length paths routing language.



(a) Clique topology

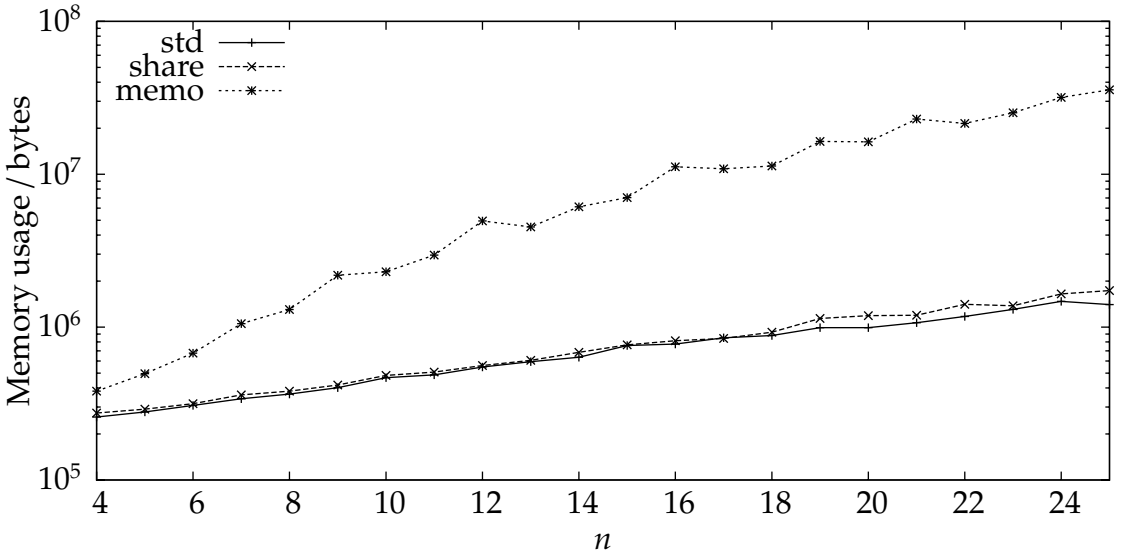


(b) Ring topology

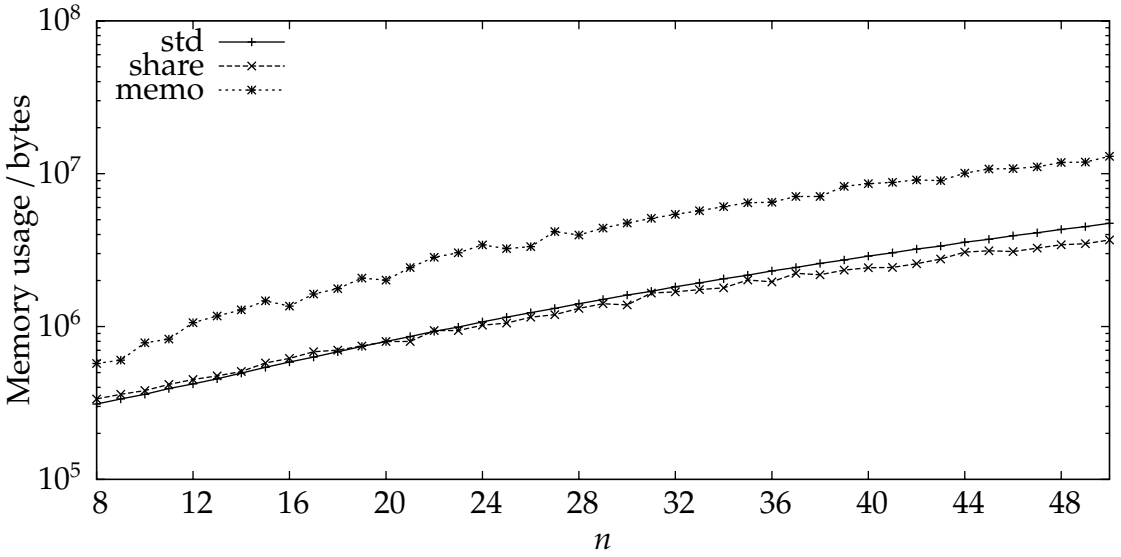


(c) Grid topology

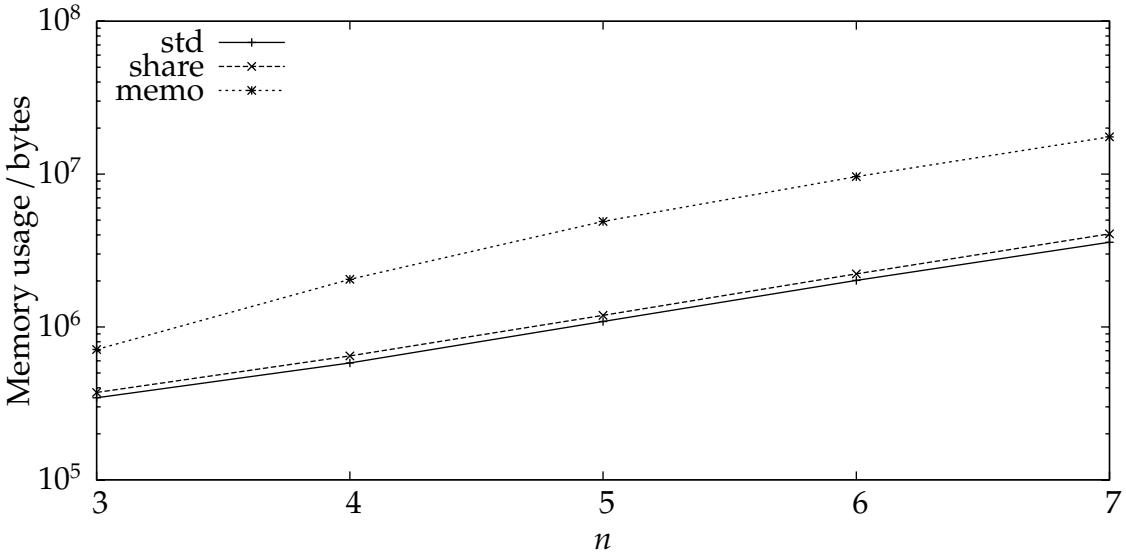
Figure 8.8: CPU usage for k-best paths routing language.



(a) Clique topology



(b) Ring topology



(c) Grid topology

Figure 8.9: Memory usage for k-best paths routing language.

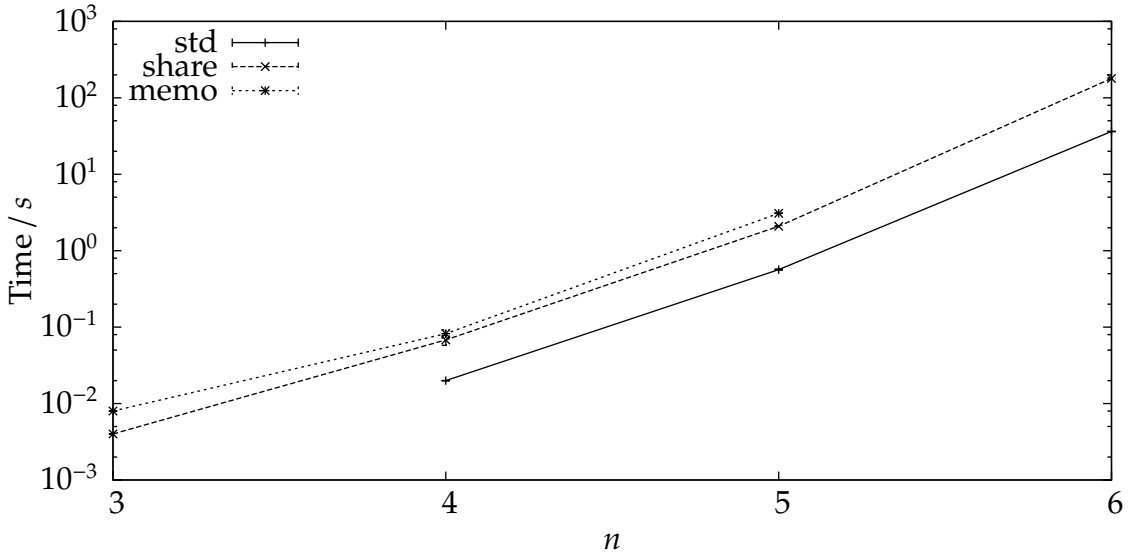
8.3.3 Martelli's algebra

The results for the Martelli routing language demonstrate that there are some routing languages upon which our current optimisations appear to have little effect (for the tested topologies). The Martelli routing language is the most expensive of the three languages in terms of both time and memory.

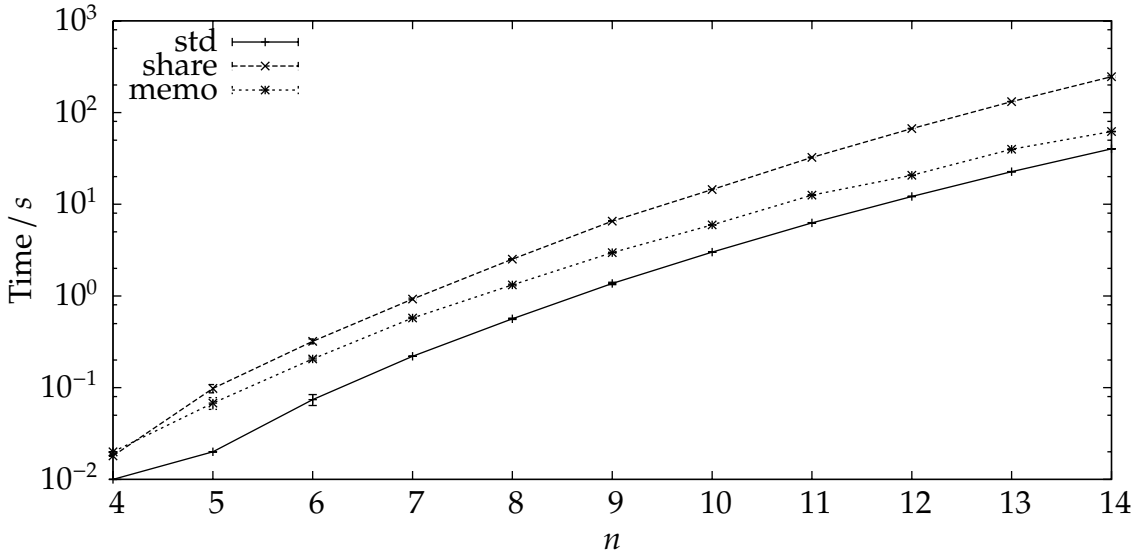
For the clique topology, there are an exponential number of cut-sets for each pair of nodes as a function of the number of nodes within the clique. Hence we are limited to testing relatively small clique sizes. We see that sharing and memoisation both lead to worse time characteristics (Figure 8.10(a)). However, sharing gives a slight benefit for memory usage at $n = 6$, as shown in Figure 8.11(a). The memory requirements for memoisation exceeded the capabilities of the test machine at $n = 6$. We hypothesise that this is due to the construction of a large number of intermediate sets.

The ring topology has order $O(n^2)$ cut-sets for each pair of nodes, and therefore the routing language can be tested with a greater number of nodes than for the clique topology. Whilst sharing and memoisation still lead to larger times (Figure 8.10(b)), we see that sharing has an appreciable benefit on memory usage (Figure 8.11(b)). This may be due to a large proportion of identical cut-sets for different pairs of nodes.

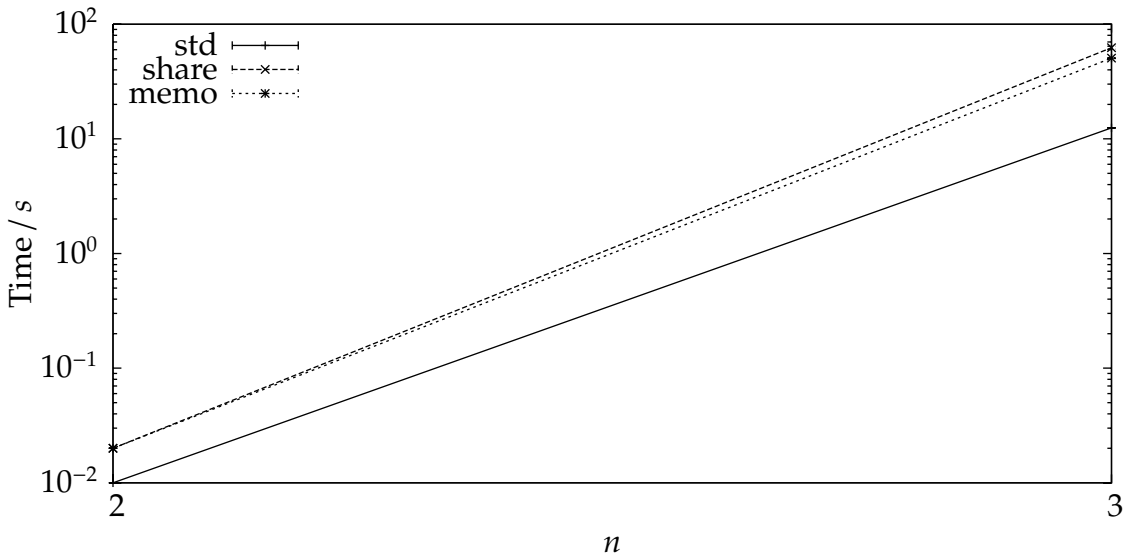
Finally, the grid topology presents largely similar time and memory characteristics to the clique topology. The timing performance is shown in Figure 8.10(c) and the the memory performance is shown in Figure 8.11(c). In common with the other two topologies, there is a reduction in memory usage for the sharing optimisation.



(a) Clique topology

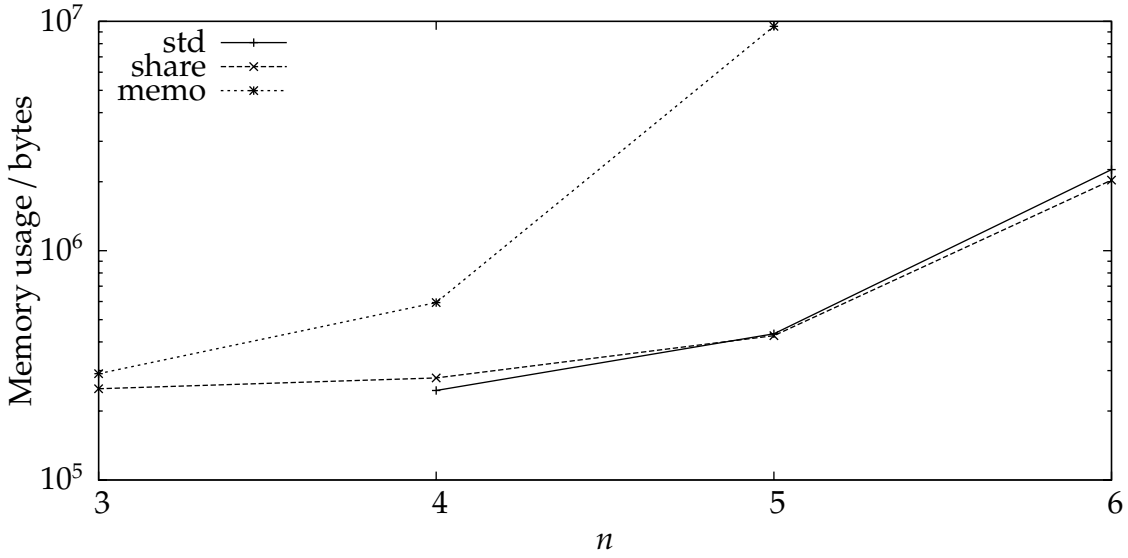


(b) Ring topology

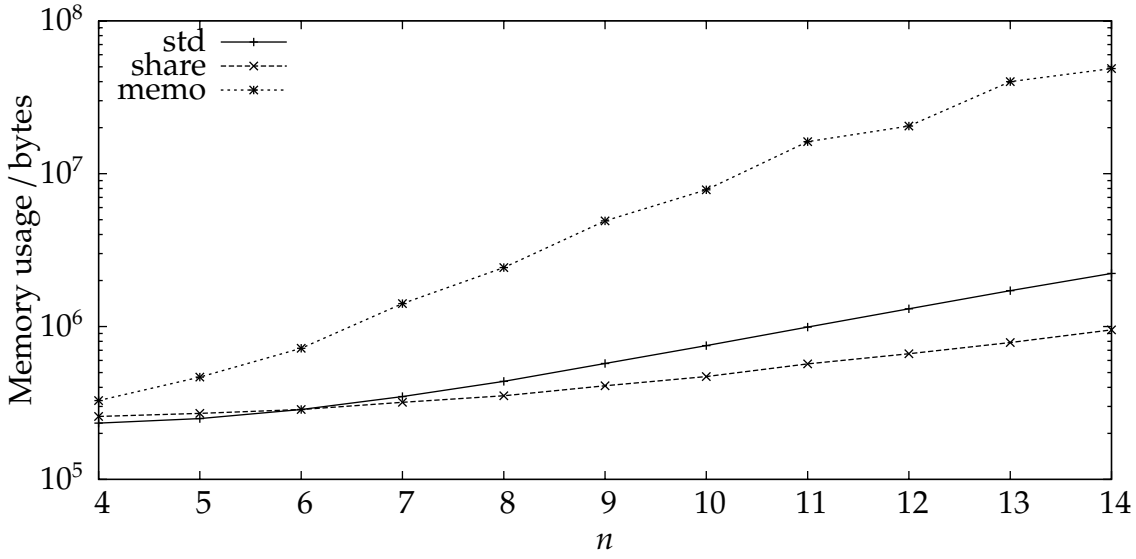


(c) Grid topology

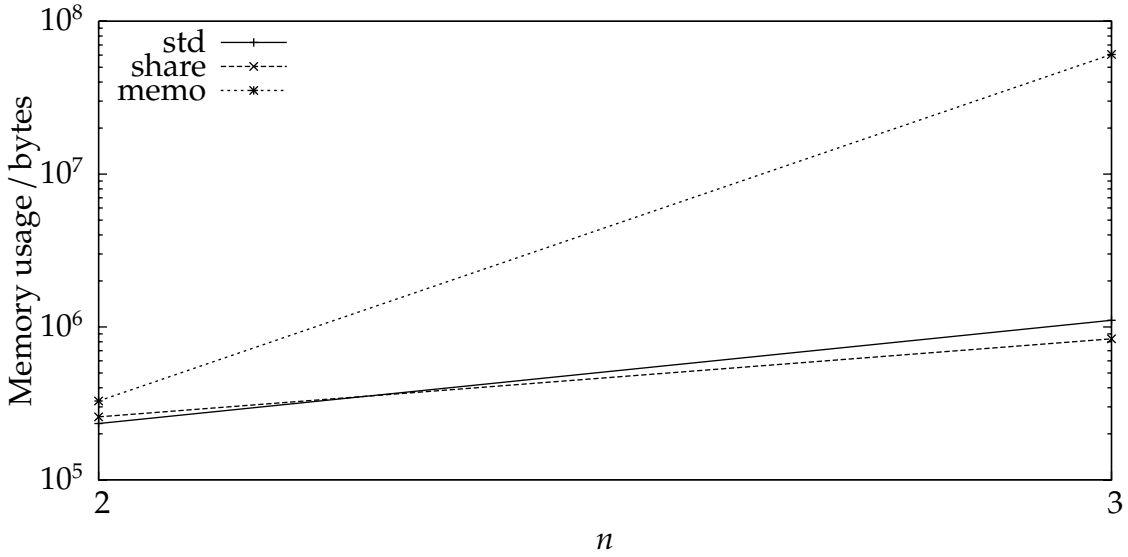
Figure 8.10: CPU usage for Martelli routing language.



(a) Clique topology



(b) Ring topology



(c) Grid topology

Figure 8.11: Memory usage for Martelli routing language.

8.4 Discussion

8.4.1 Relating results to optimisation techniques

We have demonstrated that in many cases it is possible to substantially optimise the example routing languages in the offline case, although the exact benefits are dependent upon the particular topology. We summarise the results below.

Sharing can give a small benefit in terms of execution time e.g. for the k-best paths on the ring and grid topologies. However, the benefit is very much dependent upon the particular routing language and topology. For example, sharing introduces a small time penalty for the minimum-length paths routing language. The main benefit from sharing appears to be the decrease in memory usage. For example, memory usage is halved on the ring topology for the k-best paths at $n = 50$ (Figure 8.9(b)).

Memoisation can give large time improvements at the expense of large increases in memory usage. For example, memoisation of the k-best paths routing language on the grid topology at $n = 7$ has an approximate order-of-magnitude time improvement over the standard variant (Figure 8.8(c)). However, this is associated with an approximate order of magnitude increase in memory usage. (Figure 8.9(c)). We believe that it may be possible to decrease the memory usage by reclaiming memoised entries after a period of inactivity.

Turning to the 'no_min' algebraic optimisation, we see that avoiding minimisation can bring large benefits, although again this is dependent upon the particular topology. For example, this optimisation applied to the minimum-length paths routing language on the grid topology at $n = 6$ approximately halves the execution time of the 'memo' optimisation alone (Figure 8.6(c)). This optimisation can also reduce the number of values that require caching, hence decreasing the memory usage for memoisation. On the same test, this optimisation approximately halves memory usage of the 'memo' optimisation alone (Figure 8.7(c)).

The Martelli routing language demonstrates that not all algebras are currently amenable to optimisation. We are only able to obtain a minimal memory performance increase using the 'share' optimisation. Execution times increase for all optimisations. Research suggests that it may be possible to apply algebraic optimisations to this algebra [121, 122], although this remains a topic for future work.

8.4.2 Applicability of results to online routing

We now place the results of this chapter in the context of online routing. We aim to give some indication of the applicability of these results to the performance of online routing algorithms that might be developed using the metarouting system.

We first make the distinction between the number of steps required to reach convergence and the complexity of the operations within each step. In Chapter 2 we discussed how under certain conditions it is possible bound the number of steps required for convergence of routing algorithms. For example, the generalised Bellman-Ford algorithm requires $O(|V|^3)$ steps providing that the algebra is increasing (§ 2.7.1). However, the complexity of each step within an algorithm is dependent upon the particular algebra. For example, Martelli's algebra requires $O(n^2)$ comparisons in the worst case to minimise a set with n elements.

All of the performance experiments presented in this chapter have used the offline matrix algorithm. We now discuss the change in complexity when moving from an offline algorithm to an online algorithm, as might be found in a deployed routing system. In general, the convergence time of an online algorithm may be degraded by a number of factors, including the arrival order of messages and topology changes. For example, the distributed Bellman-Ford algorithm may take an exponential number of steps to converge, and can also exhibit counting to infinity behaviour. Furthermore, there is a decrease in sharing for online algorithms because metrics are distributed between multiple routing daemons. This may reduce the benefits of sharing-related optimisations, such as hash-consing. Accurately quantifying the complexity of an online routing algorithm remains a current research topic.

Finally, we discuss the additional overhead imposed by re-implementing an existing online routing protocol within the metarouting system. Philip Taylor has conducted some initial experiments in this area, focusing upon BGP. The experiments compared the performance of the generalised Quagga BGP algorithm with a BGP-like routing algebra to the standard Quagga BGP routing protocol. The time and memory usage for each protocol was measured whilst processing routing entries from a publicly available BGP routing table dump. An approximate ten percent decrease in execution speed was observed for the generalised BGP algorithm when compared to the standard BGP routing protocol. There was also a slight decrease in memory usage for the generalised BGP algorithm, although this was perhaps due to the fact that the associated BGP-like routing algebra used a simplified AS path representation. Additional experiments are needed to more accurately quantify the performance changes when re-implementing current routing protocols within the metarouting system.

Chapter 9

Deriving forwarding paths from routing solutions

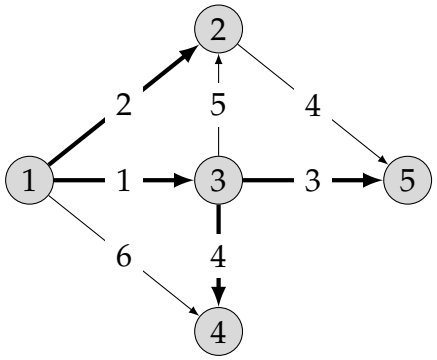
In this section we clarify the distinction between routing and forwarding, and demonstrate several methods for obtaining forwarding paths from routing solutions. We take a high-level approach which ignores implementation details. Our approach is to use an algebraic construction known as a semimodule. Whilst this is currently an abstract model, future work involves integrating it with the metarouting system.

We commence by discussing the differences between routing and forwarding (§ 9.1). We then show how to construct forwarding tables from routing tables by *importing* destinations that are external to the routing domain (§ 9.2). Next, we generalise this model using semimodules (§ 9.3). These structures allow us to model common Internet routing idioms such as *hot-potato* and *cold-potato* routing. We then show how to use our formalism to model OSPF forwarding (§ 9.4). We conclude by considering the case in which the semiring or semimodule distributivity laws no longer hold (§ 9.5).

Both this chapter and Chapter 10 are based upon joint work with Timothy Griffin that is published in [124]. However, the specific text of these chapters is the author's own work.

9.1 Introduction

Recall from Chapter 1 that routers use routing protocols to dynamically compute *routing tables*. The data from routing tables is then used to automatically construct *forwarding tables*, which control the paths that datagrams follow as they traverse the network. We consider routing to be a function that computes paths within a specified routing domain, whilst forwarding determines how these paths are actually used to carry traffic.



(a) Labelled graph

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & 2 & 1 & 6 & \infty \\ \infty & \infty & \infty & \infty & 4 \\ \infty & 5 & \infty & 4 & 3 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \end{array}$$

(b) Adjacency matrix

$$\mathbf{R} = \mathbf{A}^* = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 5 & 4 \\ \infty & 0 & \infty & \infty & 4 \\ \infty & 5 & 0 & 4 & 3 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(c) Routing matrix

Figure 9.1: Example of algebraic routing using the MinPlus semiring

Routing and forwarding have similar roles, and the terms are often used interchangeably. In the simplest case, the paths that are used for forwarding are exactly those that have been computed by routing. However, even in this case we see a distinction; forwarding must select between paths that have equal cost to the same destination. Possible choices include randomly choosing a path or load-balancing traffic between several equal-cost paths.

In this chapter we model the *infrastructure* of a network as a directed graph $G = (V, E)$. Given a pair of nodes $i,\ j \in V$, routing computes a set of paths in G that can be used to transit data from i to j. We algebraically model routing tables using matrix semirings. A network-wide routing table is represented as a $V \times V$ *routing matrix* $\mathbf{R}$ that satisfies the equation

$$\mathbf{R} = (\mathbf{A} \otimes \mathbf{R}) \oplus \mathbf{I},$$

where $\mathbf{A}$ is the adjacency matrix induced by a graph labelled over a semiring S. Each matrix entry $\mathbf{R}(i,\ j)$ in fact corresponds to the minimal-cost path *weight* from i to j, which is implicitly associated with a set of optimal paths from node i to node j. Providing that the semiring is selective, then it is straightforward to recover the associated paths. In Chapter 2 we described how various algorithms may be used to compute the routing matrix $\mathbf{R} = \mathbf{A}^*$ from the adjacency matrix.

We use the example in Figure 9.1 to briefly recap the algebraic approach to the construction of routing matrices. Figure 9.1(a) presents a simple five node graph with integer labels and Figure 9.1(b) shows the associated adjacency matrix. We wish to compute *shortest-distances* between each pair of nodes and therefore we compute the

closure using the semiring MinPlus = ($\mathbb{N}^\infty$, min, +), The resulting matrix is given in Figure 9.1(c). It is straightforward to recover the corresponding paths because MinPlus is *selective*. That is, for all $x, y \in S$ we have $x \oplus y \in \{x, y\}$. Hence the computed weights actually correspond to the weights of individual paths. The bold arrows in Figure 9.1(a) denote the shortest-paths tree rooted at node 1; the corresponding path weights are given in the first row of the matrix in Figure 9.1(c).

In this chapter we build upon this model by assuming that there is a set of *external destinations* D that are independent of the network. Destinations can be *directly attached* to any number of nodes in the network. We model the attachment information using a $V \times D$ *mapping matrix* $\mathbf{M}$. The entry $\mathbf{M}(i, d)$ contain a cost associated with the attachment of external destination d to infrastructure node i.

Forwarding then consists of finding minimal-cost paths from nodes $i \in V$ to destinations $d \in D$. We model forwarding using a $V \times D$ *forwarding matrix* $\mathbf{F}$, where each entry $\mathbf{F}(i, d)$ is implicitly associated with a set of paths from node i to destination d. We treat the construction of $\mathbf{F}$ as the process of solving the equation

$$\mathbf{F} = (\mathbf{A} \triangleright \mathbf{F}) \square \mathbf{M},$$

where $\mathbf{F}$ and $\mathbf{M}$ contain entries in a semimodule $N = (N, \square, \triangleright)$ over the semiring S. We present several semimodule constructions that model common Internet forwarding idioms such as *hot-potato* and *cold-potato* forwarding. Each such method potentially leads to a different set of forwarding paths; the examples in this chapter all share the same routing matrix, yet have distinct forwarding paths.

In Chapter 10 we again extend this model to show how mapping tables can themselves be generated from forwarding tables. This provides a model of a simple type of *route redistribution* between distinct routing protocols.

9.2 Attaching destinations

As before, suppose that our network is represented by the graph $G = (V, E)$, labelled with elements from the semiring S. Let the external nodes be chosen from some set D, satisfying $V \cap D = \emptyset$. Attach external nodes to G using the *attachment edges* $E' \subseteq V \times D$. In the simplest case, the edges in E' have weights from S, although in the next section (§ 9.3) we show how to relax this assumption. Let the $V \times D$ *mapping matrix* $\mathbf{M}$ represent the attachment edges.

We now wish to compute the $V \times D$ matrix $\mathbf{F}$ of shortest-path weights from nodes in V to nodes in D. We term $\mathbf{F}$ a *forwarding solution* because it comprises the information required to reach destinations, instead of other infrastructure nodes. We compute $\mathbf{F}$ by

post-multiplying the routing solution $\mathbf{R}$ by the mapping matrix $\mathbf{M}$. That is, for $i \in V$ and $d \in D$, we have

$$\begin{aligned}\mathbf{F}(i, d) &= (\mathbf{R} \otimes \mathbf{M})(i, d) \\ &= \bigoplus_{k \in V} \mathbf{R}(i, k) \otimes \mathbf{M}(k, d) \\ &= \bigoplus_{k \in V} \delta(i, k) \otimes \mathbf{M}(k, d).\end{aligned}$$

Hence we see that $\mathbf{F}(i, d)$ corresponds to the shortest total path length from i to d. In other words, $\mathbf{F}$ solves the *forwarding equation*

$$\mathbf{F} = (\mathbf{A} \otimes \mathbf{F}) \oplus \mathbf{M}.$$

Note that we are able to change the value of $\mathbf{M}$ and recompute $\mathbf{F}$ without recomputing $\mathbf{R}$. From an Internet routing perspective this is an important property; if the external information is dynamically computed (by another routing protocol, for example) then it may frequently change, and in such instances it is desirable to avoid recomputing routing solutions.

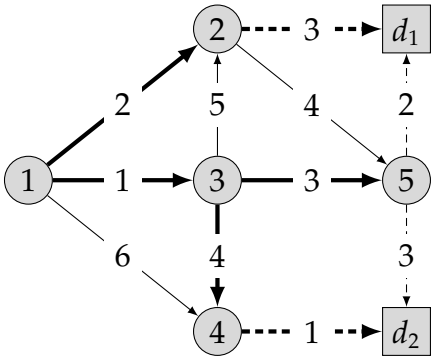
We illustrate this model of forwarding in Figure 9.2. The labelled graph of Figure 9.2(a) is based upon that in Figure 9.1(a), with the addition of two external nodes: d_1 and d_2. The adjacency matrix $\mathbf{A}$ remains as before, whilst the mapping matrix $\mathbf{M}$, given in Figure 9.2(b), contains the attachment information for d_1 and d_2. The forwarding solution $\mathbf{F}$ that results from multiplying $\mathbf{R}$ by $\mathbf{M}$ is given in Figure 9.2(c). Again, it is easy to verify that the elements of $\mathbf{F}$ do indeed correspond to the weights of the shortest paths from nodes in V to nodes in D.

9.3 Generalised attachment

Within Internet routing, it is common for the entries in routing and forwarding tables to have distinct types, and for these types to be associated with distinct order relations. We therefore generalise the import model of the previous section to allow this possibility. In particular, we show how to solve this problem using algebraic structures known as *semimodules*.

9.3.1 Semimodules

A (left) semimodule is an algebraic structure $N = (N, \square, \triangleright)$ that is defined over a semiring $S = (S, \oplus, \otimes)$. The operator $\square \in N \times N \to N$ is used to summarise elements of N, whilst the operator $\triangleright \in S \times N \to N$ can be viewed as a method of 'lifting' elements from S into N.



(a) Graph with external information

$$\mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} \infty & \infty \\ 3 & \infty \\ \infty & \infty \\ \infty & 1 \\ 2 & 3 \end{bmatrix} \end{array}$$

(b) Mapping matrix

$$\mathbf{R} \otimes \mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} 5 & 6 \\ 3 & 7 \\ 5 & 5 \\ 9 & 1 \\ 2 & 3 \end{bmatrix} \end{array}$$

(c) Forwarding matrix

Figure 9.2: Example of combining routing and mapping matrices to create a *forwarding* matrix

Analogously to a semiring, a semimodule must satisfy several algebraic properties. Firstly, we require that $(N, \square)$ is a commutative semigroup. Secondly, the operators must satisfy the distributivity laws

$$\begin{aligned} x \triangleright (m \square n) &= (x \triangleright m) \square (x \triangleright n), \\ (x \oplus y) \triangleright m &= (x \triangleright m) \square (y \triangleright m), \end{aligned}$$

where $x, y \in S$ and $m, n \in N$. Thirdly, the semigroup multiplicative identity must satisfy the property $\alpha_{\otimes} \triangleright m = m$. Finally, we require the existence of an identity for $\square$, which we denote as $\alpha_{\square}$. This value must satisfy the properties $\alpha_{\oplus} \triangleright m = \alpha_{\square}$, and $x \triangleright \alpha_{\square} = \alpha_{\square}$. In our applications, $\square$ is often idempotent.

A right semimodule $N = (N, \square, \triangleleft)$ instead has an operator $\triangleleft \in N \times S \to N$, and must satisfy 'right' versions of the previous properties. We note that a semimodule can be seen as an instance of a semigroup transform (Chapter 7), where the label type is S. However, a semimodule has additional algebraic structure that is not required of a semigroup transform. For example, there is an 'internal' $\oplus$ operator.

We now show our use of a semimodule is a natural consequence of combining routing and mapping entries of different types.

9.3.2 Using semimodules for forwarding

Assume that we are using the semiring $S = (S, \oplus, \otimes)$ and suppose that we wish to construct forwarding matrices with elements from the idempotent, commutative semi-

group $N = (N, \square)$. Furthermore, suppose that the mapping matrix $\mathbf{M}$ contains entries over N. In order to compute forwarding entries, it is necessary to combine routing entries with mapping entries, as before. However, we can no longer use the multiplicative operator from S because the mapping entries are of a different type. Therefore we introduce an operator $\triangleright \in (S \times N) \to N$ for this purpose. We can now construct forwarding entries as

$$\begin{aligned} \mathbf{F}(i, d) &= (\mathbf{R} \triangleright \mathbf{M})(i, d) \\ &= \bigsquare_{k \in V} \mathbf{R}(i, k) \triangleright \mathbf{M}(k, d). \end{aligned} \tag{9.1}$$

It is also possible to equationally characterise the resulting forwarding entries, as before. Assume that $\mathbf{R}$ is a routing solution i.e. it satisfies the routing equation

$$\mathbf{R} = (\mathbf{A} \otimes \mathbf{R}) \oplus \mathbf{I}.$$

Then, providing that the algebraic structure $N = (N, \square, \triangleright)$ is a semimodule, $\mathbf{F} = \mathbf{R} \triangleright \mathbf{M}$ is a solution to the forwarding equation

$$\mathbf{F} = (\mathbf{A} \triangleright \mathbf{F}) \square \mathbf{M}.$$

Substituting $\mathbf{F} = \mathbf{A}^* \triangleright \mathbf{M}$ into the left-hand side of this equation, we obtain

$$\begin{aligned} & (\mathbf{A} \triangleright (\mathbf{A}^* \triangleright \mathbf{M})) \square \mathbf{M} \\ = \; & ((\mathbf{A} \otimes \mathbf{A}^*) \triangleright \mathbf{M}) \square \mathbf{M} \\ = \; & ((\mathbf{A} \otimes \mathbf{A}^*) \oplus \mathbf{I}) \triangleright \mathbf{M} \\ = \; & \mathbf{A}^* \triangleright \mathbf{M} \end{aligned}$$

In other words, we can solve for $\mathbf{F}$ with $\mathbf{F} = \mathbf{A}^* \triangleright \mathbf{M}$. Significantly, we are able to use semimodules to model the mapping information whilst still retaining a semiring model of routing.

9.3.3 Hot-potato semimodule

We now develop two important semimodule constructions that model the most common manner in which routing and mapping are combined: the *hot-potato* and *cold-potato* semimodules. First define an *egress* node for a destination d as a node k within the routing domain that is directly attached to d. Hot-potato forwarding to d first selects paths to the closest egress nodes for d and then breaks ties using the mapping information. In contrast, cold-potato forwarding first selects paths to the egress nodes for d with the most preferred mapping values, and then breaks ties using the routing distances.

We now formally define the hot-potato semimodule. Let $S = (S, \oplus_S, \otimes_S)$ be an idempotent semiring with $(S, \oplus_S)$ selective and let $T = (T, \oplus_T)$ be a monoid. The hot-potato semimodule over S is defined as

$$\mathrm{Hot}(S, T) = ((S \times T) \cup \{\infty\}, \vec{\oplus}, \triangleright_{\mathrm{fst}}),$$

where $s_1 \rhd_{\text{fst}} (s,\, t) = (s_1 \otimes_S s,\, t)$, $s_1 \rhd_{\text{fst}} \infty = \infty$ and $((S \times T) \cup \{\infty\},\, \vec{\oplus})$ is the *left* lexicographic product semigroup. In common with semirings, we can lift semimodules to operate over (non-square) matrices. When lifting the hot-potato semimodule, we rename the multiplicative operator from $\rhd_{\text{fst}}$ to $\rhd_{\text{hp}}$. This is because the lifted multiplicative operator no longer simply applies its left argument to the first component of its right argument. In fact, we shall shortly see that the cold-potato semimodule uses the same underlying multiplicative operator but with a different additive operator, and therefore has a different lifted multiplicative operator.

The behaviour of the hot-potato semimodule can be algebraically characterised as follows. Suppose that for all $j \in V$ and $d \in D$, $\mathbf{M}(j,\, d) \in \{(1_S,\, t),\, \infty_T\}$ where 1_S is the multiplicative identity for S and t is some element of T. Then from Equation 9.1 it is easy to check that

$$(\mathbf{R} \rhd_{\text{hp}} \mathbf{M})(i,\, d) = \vec{\bigoplus_{\substack{j \in V \\ M(j,d) = (1_S,\, t)}}} (\mathbf{R}(i,\, j),\, t).$$

That is, as desired, the mapping metric is simply used to tie-break over otherwise minimal-weight paths to the edge of the routing domain.

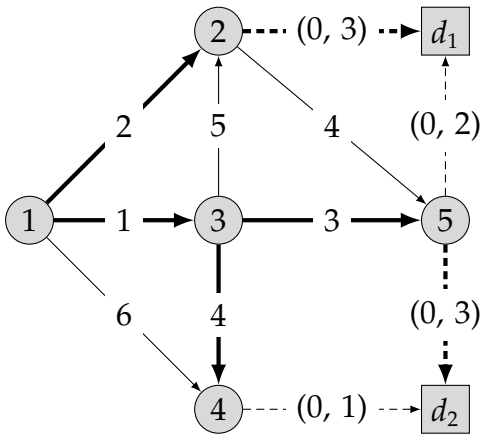
We illustrate the hot-potato model of forwarding in Figure 9.3. This example uses the semimodule Hot(MinPlus, Min), where Min = $(\mathbb{N}^\infty,\, \min)$. The graph of Figure 9.3(a) is identical to that of Figure 9.2(a), but now the attachment arcs of d_1 and d_2 are weighted with elements of the hot-potato semimodule. The associated mapping matrix is given in Figure 9.3(b), whilst the resulting forwarding table is shown in Figure 9.3(c). Note that 0 is the multiplicative identity of the MinPlus semiring. Comparing this example to Figure 9.2, we see that node 1 reaches d_2 via egress node 5 instead of node 4. This is because the mapping information is only used for tie-breaking, instead of being directly combined with the routing distance. Also, in this particular example, it is never the case that there are multiple paths of minimum cost to egress nodes, and therefore no tie-breaking is performed by the mapping information.

9.3.4 Cold-potato semimodule

Turning to cold-potato forwarding, the associated semimodule again combines routing and attachment information using the lexicographic product, but with priority now given to the attachment component. As before, let $S = (S,\, \oplus_S,\, \otimes_S)$ be a semiring and $T = (T,\, \oplus_T)$ be a monoid, but now with T idempotent and selective. The cold-potato semimodule over S is defined as

$$\text{Cold}(S,\, T) = ((S \times T) \cup \{\infty\},\, \overleftarrow{\oplus},\, \rhd_{\text{fst}}).$$

Note that $(S \times T,\, \overleftarrow{\oplus})$ the *right* lexicographic product semigroup. Again, when lifting the cold-potato semimodule to operate over matrices we rename the multiplicative



(a) Graph with external information

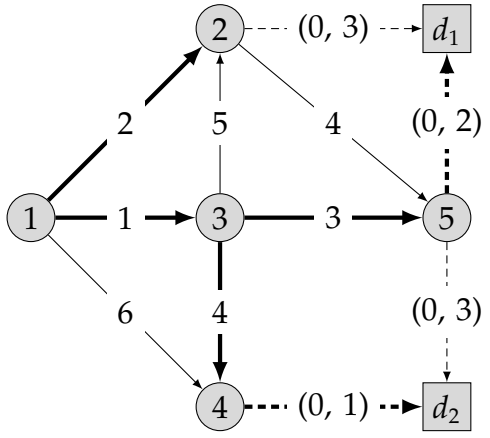
$$\mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} \infty & \infty \\ (0,3) & \infty \\ \infty & \infty \\ \infty & (0,1) \\ (0,2) & (0,3) \end{bmatrix} \end{array}$$

(b) Mapping matrix

$$\mathbf{R} \rhd_{\mathrm{hp}} \mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} (2,3) & (4,3) \\ (0,3) & (4,3) \\ (3,2) & (3,3) \\ (7,2) & (0,1) \\ (0,2) & (0,3) \end{bmatrix} \end{array}$$

(c) Forwarding matrix

Figure 9.3: Example of *hot-potato* forwarding



(a) Graph with external information

$$\mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} \infty & \infty \\ (0,3) & \infty \\ \infty & \infty \\ \infty & (0,1) \\ (0,2) & (0,3) \end{bmatrix} \end{array}$$

(b) Mapping matrix

$$\mathbf{R} \rhd_{\mathrm{cp}} \mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} (4,2) & (5,1) \\ (4,2) & (9,1) \\ (3,2) & (4,1) \\ (7,2) & (0,1) \\ (0,2) & (7,1) \end{bmatrix} \end{array}$$

(c) Forwarding matrix

Figure 9.4: Example of *cold-potato* forwarding

operator from $\rhd_{\mathrm{fst}}$ to $\rhd_{\mathrm{cp}}$.

Figure 9.4 illustrates the cold-potato model of forwarding. This example uses the cold-potato semimodule Cold(MinPlus, Min), but is otherwise identical to Figure 9.3. It is easy to verify that priority is now given to the mapping information when selecting egress nodes.

Within Internet routing, hot-potato forwarding corresponds to choosing the closest

egress point from a given routing domain. This is the default behaviour for the Border Gateway Protocol (BGP) routing protocol [9] because it tends to minimise resource usage for outbound traffic within the domain. In contrast, cold-potato forwarding allows the mapping facility to select egress nodes, and hence can lead to longer paths being chosen within the domain. As a result, cold-potato forwarding is less commonly observed in general on the Internet. However, one specific use is within client-provider peering relations in order to minimise the use of the client's network resources for inbound traffic (at the possible expense of increased resource usage on the provider's network). This is precisely the reason that the BGP MED attribute is often used (§ 1.4.2).

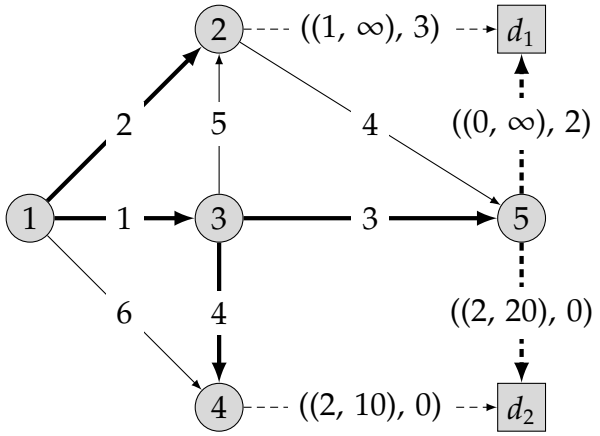
9.4 Modelling OSPF

In this section we use our semimodule model of attachment to abstractly describe how the OSPF routing protocol [125] constructs forwarding tables. We demonstrate that this process involves *multiple* mapping tables, with one for each different type of destination. The tables are then combined using a variant of the cold-potato semimodule to produce a single forwarding table. Note that for simplicity we ignore OSPF areas.

OSPF makes the distinction between routing and mapping, representing the corresponding tables as different structures within routing updates. This is in marked contrast to routing protocols such as BGP and RIP, where routing and mapping information are amalgamated within routing updates. The separated routing and mapping of OSPF permits a significant amount of additional policy control when computing forwarding tables to external destinations.

In fact, OSPF defines *three* different kinds of destinations, each of which can be modelled using a different mapping table. The first type, which we term *type 0*, corresponds to destinations that are directly attached to an OSPF router. The second and third types, we are termed *type 1* and *type 2* respectively [11], correspond to different ways of attaching external destinations. OSPF strictly prefers type 0 destinations to type 1 destinations, and type 1 destinations to type 2 destinations. Type 2 destinations may additionally be associated with another, statically-specified metric. This latter metric is given priority to the computed metric, giving rise to a form of cold-potato forwarding. In this section we generalise the static metric by assuming that it may be modelled using a commutative, idempotent monoid, $U = (U, \oplus_U)$.

Building upon our previous examples, we present an example network in Figure 9.5(a). We use this example to illustrate our model of OSPF forwarding. Here we have that U is a bandwidth metric. In particular, $U = (\mathbb{N}^\infty, \max)$. We note that the annihilator for this monoid is $\omega_U = 0$.



(a) Graph with external information

$$\mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} \infty & \infty \\ ((1,\infty),3) & \infty \\ \infty & \infty \\ \infty & ((2,10),0) \\ ((0,\infty),2) & ((2,20),0) \end{bmatrix} \end{array}$$

(b) Mapping matrix

$$\mathbf{R} \triangleright_{\text{cp}} \mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} ((0,\infty),6) & ((2,20),4) \\ ((0,\infty),6) & ((2,20),8) \\ ((0,\infty),5) & ((2,20),3) \\ ((0,\infty),9) & ((2,20),7) \\ ((0,\infty),2) & ((2,20),0) \end{bmatrix} \end{array}$$

(c) Forwarding matrix

Figure 9.5: Example of *idealised-OSPF* forwarding

Our model uses the following set, into which we embed each destination type:

$$W = (\{0, 1, 2\} \times U) \times \mathbb{N}^{\infty}) \cup \{\infty\}.$$

We define the embedding into W as follows:

Type	Metric	Embedding
0	$m \neq \infty$	$((0, \omega_U), m)$
1	$m \neq \infty$	$((1, \omega_U), m)$
2	$u \neq \omega_U$	$((2, u), 0)$

The elements of W are ordered using the right lexicographic product

$$(W, \vec{\oplus}) = (\{0, 1, 2\}, \min) \vec{\times} (U, \oplus_U) \vec{\times} (\mathbb{N}^{\infty}, \min).$$

This order does not alter the pre-existing preference between metrics of the same destination type. However, for destinations of *different* types, the order is defined so as to respect the preference defined within the OSPF specification i.e. type 0 destinations are strictly preferred to type 1 destinations, etc. We then represent each destination type t as a separate mapping matrix $\mathbf{M}_t$. Whilst we do not discuss how these matrices might be constructed, it is straightforward to imagine how this might be done. For our

example in Figure 9.5(a), we obtain the following mapping matrices:

$$\mathbf{M}_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \left[\begin{array}{cc} \infty & \infty \\ \infty & \infty \\ \infty & \infty \\ \infty & \infty \\ ((0,\infty),2) & \infty \end{array}\right] \end{array} \quad \mathbf{M}_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \left[\begin{array}{cc} \infty & \infty \\ ((1,\infty),3) & \infty \\ \infty & \infty \\ \infty & \infty \\ ((1,\infty),17) & \infty \end{array}\right] \end{array} \quad \mathbf{M}_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \left[\begin{array}{cc} \infty & \infty \\ ((2,40),0) & \infty \\ \infty & \infty \\ \infty & ((2,10),0) \\ ((2,10),0) & ((2,20),0) \end{array}\right] \end{array}$$

We then construct a single mapping matrix $\mathbf{M}$ by combining the individual mapping matrices using the order $(W, \vec{\oplus})$:

$$\mathbf{M} = \mathbf{M}_0 \,\vec{\oplus}\, \mathbf{M}_1 \,\vec{\oplus}\, \mathbf{M}_2.$$

Returning to our example, the resulting mapping matrix is shown in Figure 9.5(b). Finally, we define the OSPF semimodule as

$$\mathrm{OSPF}(U) = (W, \vec{\oplus}, \triangleright_{\mathrm{snd}})$$

where

$$\begin{aligned} m \triangleright_{\mathrm{snd}} ((l, u), m') &= ((l, u), m + m') \\ m \triangleright_{\mathrm{snd}} \infty &= \infty. \end{aligned}$$

This structure is a variant of the cold-potato semimodule; here, instead of combining routing data with the first component of the mapping information and using the right lexicographic order, we instead combine it with the second component and use the left lexicographic order. Hence we refer to the lifted multiplicative operator as $\triangleright_{\mathrm{cp}}$.

We then construct the OSPF forwarding matrix as $\mathbf{F} = \mathbf{R} \triangleright_{\mathrm{cp}} \mathbf{M}$. The forwarding matrix for our example is illustrated in Figure 9.5(c). For destination d_1, we see that the type 0 route is given priority over the type 1 route. In contrast, there are two type 2 routes for destination d_2, and hence the bandwidth component is used as a tie-breaker.

9.5 The non-distributive case

In Section 9.3.2 we demonstrated how to directly solve the forwarding equation $\mathbf{F} = (\mathbf{A} \triangleright \mathbf{F}) \square \mathbf{M}$. However, in some cases it may also be possible to solve this equation *iteratively*. That is, we define a sequence of approximations

$$\begin{aligned} \mathbf{F}^{[0]} &= \mathbf{M}, \\ \mathbf{F}^{[k+1]} &= (\mathbf{A} \triangleright \mathbf{F}^{[k]}) \square \mathbf{M}. \end{aligned}$$

In the case that $\mathbf{A}^*$ exists, and $(N, \square, \triangleright)$ is a semimodule over S, then we can show that $\lim_{k\to\infty} \mathbf{F}^{[k]}$ exists and that it has the value $\mathbf{A}^* \triangleright \mathbf{M}$.

Two problems may be encountered when attempting to use the iterative solution method upon models of Internet routing protocols. The first problem is that S may not be a semiring due to the semiring distributivity laws not holding. However, even in such a situation, it may be possible to obtain a *locally optimal* solution using the iterative method [91]. Now suppose that S *is* a semiring, but that the semimodule distributivity axioms do not hold. Again, in such a situation it may be possible to extend the work of [91] to show that the iterative method outlined above does indeed converge to locally optimal solution. Both of these areas are topics for future work.

Chapter 10

Simple route redistribution

In this chapter we describe a mechanism for the interaction of routing protocols. We term this facility *simple route redistribution* after the mechanism for protocol interaction found in current routers. We build the redistribution model using the attachment model from Chapter 9. We commence by introducing simple route redistribution (§ 10.1), before describing the model in detail (§ 10.2). We then conclude by relating our redistribution model to proposals for creating a more scalable routing and addressing system for the Internet (§ 10.3).

10.1 Introduction

Recall from Chapter 1 that route redistribution is a mechanism for copying routes between RIBs. It is often used for 'joining' routing protocols, such as when merging pairs of networks. Redistribution can also increase resilience by permitting paths to be used within an alternative routing domain. Route redistribution is a *local* mechanism; it is enabled on individual routers. However, route redistribution has *global* effect; redistributing routes from routing domain A to routing domain B may affect route selection for any router within B.

In Section 1.3 we discussed the problems that are associated with current route redistribution mechanisms; principally, route redistribution can cause routing oscillations and forwarding loops [24, 27, 28, 29]. Detecting whether a given set of configurations is subject to unsafe behaviour is NP-hard. This has led to the creation of informal sets of guidelines [27] for safely configuring route redistribution.

In this chapter we take a different approach to the problem of safe redistribution. In contrast to current approaches that attempt to understand the low-level details of redistribution [28], we adopt a top-down model. This leads us to consider redistribution as an abstraction that is built *on top of* routing protocols. Our simple route redistribution model only affects forwarding, and *not* routing. This avoids our model from

being overly entangled in low-level implementation details. Indeed, we believe that introducing this level of abstraction is essential for reasoning about the effects of redistribution and eliminating safety violations. Our model is notable in being the first algebraic account of route redistribution.

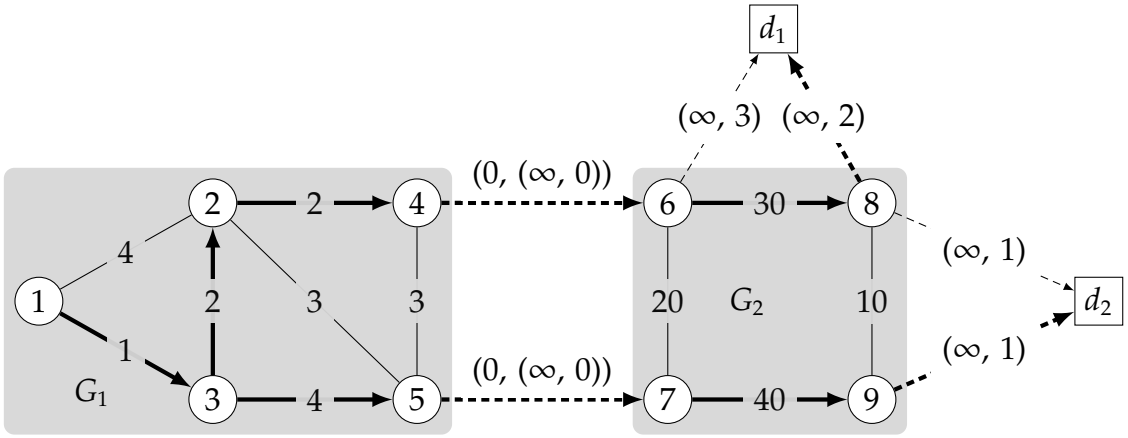
Our simple route redistribution model builds upon the attachment model of Chapter 9 to allow forwarding between multiple domains (here, we limit ourselves to modelling the case where there are two routing domains, although it is possible to generalise to a greater number). Our redistribution model uses the forwarding table from one domain to *dynamically* construct the mapping table of another. That is, when redistributing from A to B, domain B exploits the forwarding paths from A to reach the destination of A. Note that the forwarding matrices of A may themselves be constructed using semimodules, as described in Chapter 9. Our model additionally demonstrates that it is possible for each routing/forwarding domain to use a different semiring/semimodule pair.

10.2 Simple route redistribution

In this section we describe specifics of the simple route redistribution model. Begin by assuming that there are two routing domains, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Also, assume that there is a set of destinations, D, with V_1, V_2 and D pair-wise disjoint. Let G_1 be connected to G_2 with the attachment arcs $E_{1,2} \subseteq V_1 \times V_2$, represented as the $V_1 \times V_2$ *bridging matrix* $\mathbf{B}_{1,2}$. Similarly, let G_2 be connected to D with the attachment arcs $E_{2,d} \subseteq V_2 \times D$, represented as the $V_2 \times D$ attachment matrix $\mathbf{M}_2$. Let $\mathbf{F}_2$ be the forwarding matrix for G_2. We demonstrate how to construct a forwarding matrix F_1 from V_1 to D.

We use Figure 10.1 as a running example. Figure 10.1(a) illustrates two graphs, G_1 and G_2. The second graph, G_2, is directly connected to destinations d_1 and d_2 and therefore we are able to compute the forwarding matrix for G_2 using the method from Section 9. We model the routing in G_2 in the standard manner, using the bandwidth semiring MaxMin = ($\mathbb{N}^\infty$, max, min). The resulting routing matrix $\mathbf{R}_2$ is given in Figure 10.1(c). Forwarding is then modelled using the cold-potato semimodule Cold(MaxMin, Min), where Min = ($\mathbb{N}^\infty$, min). The forwarding matrix is constructed as $\mathbf{F}_2 = \mathbf{R}_2 \triangleright_{\text{cp}} \mathbf{M}_2$, where $\mathbf{M}_2$ is the mapping matrix for G_2. We give the mapping and forwarding matrices in Figures 10.1(b) and 10.1(d) respectively.

In order to compute a forwarding matrix from V_1 to D, we must first construct a mapping matrix $\mathbf{M}_1$ from V_1 to D by combining the forwarding matrix $\mathbf{F}_2$ from G_2 with the bridging matrix $\mathbf{B}_{1,2}$. Let the forwarding in G_2 be modelled using the semimodule $N_2 = (N_2, \square_2, \triangleright_2)$ and let the bridging matrix be modelled using the semigroup $(N_1, \square_1)$. We combine the forwarding and bridging information using a *right* semimod-



(a) Multiple graphs with external information

$$\mathbf{M}_2 = \begin{array}{c} \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} (\infty, 3) & \infty \\ \infty & \infty \\ (\infty, 2) & (\infty, 1) \\ \infty & (\infty, 1) \end{bmatrix} \end{array}$$

(b) G_2 mapping matrix

$$\mathbf{R}_2 = \begin{array}{c} \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 7 & 8 & 9 \end{array} \\ \begin{bmatrix} \infty & 20 & 30 & 20 \\ 20 & \infty & 20 & 40 \\ 30 & 20 & \infty & 20 \\ 20 & 40 & 20 & \infty \end{bmatrix} \end{array}$$

(c) G_2 routing matrix

$$\mathbf{F}_2 = \begin{array}{c} \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} (30, 2) & (30, 1) \\ (20, 2) & (40, 1) \\ (\infty, 2) & (\infty, 1) \\ (20, 2) & (\infty, 1) \end{bmatrix} \end{array}$$

(d) G_2 forwarding matrix

$$\mathbf{B}_{1,2} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 7 & 8 & 9 \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ (0, (\infty, 0)) & \infty & \infty & \infty \\ \infty & (0, (\infty, 0)) & \infty & \infty \end{bmatrix} \end{array}$$

(e) G_1 to G_2 bridging matrix

$$\mathbf{M}_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} \infty & \infty \\ \infty & \infty \\ \infty & \infty \\ (0, (30, 2)) & (0, (30, 1)) \\ (0, (20, 2)) & (0, (40, 1)) \end{bmatrix} \end{array}$$

(f) G_1 mapping matrix

$$\mathbf{R}_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 3 & 1 & 5 & 5 \\ 3 & 0 & 2 & 2 & 3 \\ 1 & 2 & 0 & 4 & 4 \\ 5 & 2 & 4 & 0 & 3 \\ 5 & 3 & 4 & 3 & 0 \end{bmatrix} \end{array}$$

(g) G_1 routing matrix

$$\mathbf{F}_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} (5, (30, 2)) & (5, (40, 1) \\ (2, (30, 2)) & (2, (30, 1) \\ (4, (30, 2)) & (4, (40, 1) \\ (0, (30, 2)) & (0, (30, 1) \\ (0, (20, 2)) & (0, (40, 1) \end{bmatrix} \end{array}$$

(h) G_1 forwarding matrix

Figure 10.1: Example of *simple route redistribution*

ule $(N_1, \boxplus_1, \lhd_1)$ over N_2 i.e. with $\lhd_1 \in (N_1 \times N_2) \to N_1$. Then we compute the mapping matrix from G_1 as $\mathbf{M}_1 = \mathbf{B}_{1,2} \lhd_1 \mathbf{F}_2$.

Returning to Figure 10.1, the bridging matrix $\mathbf{B}_{1,2}$ is illustrated in Figure 10.1(e). We combine the forwarding matrix $\mathbf{F}_2$ with $\mathbf{B}_{1,2}$ using the right version of the semimodule

$$\text{Hot}(\text{MinPlus}, \text{Cold}(\text{MaxMin}, \text{Min})).$$

The resulting mapping matrix $\mathbf{M}_1 = \mathbf{B}_{1,2} \lhd_{\text{hp}} \mathbf{F}_2$ is illustrated in Figure 10.1(f).

Finally, we must combine the mapping matrix $\mathbf{M}_1$ with the routing solution $\mathbf{R}_1$. Suppose that $\mathbf{R}_1$ has been computed using the semiring S. Then we construct a *left* semimodule $(N_1, \boxplus_1, \rhd_1)$ over S. Compute the forwarding matrix for G_1 as

$$\mathbf{F}_1 = \mathbf{R}_1 \rhd_1 \mathbf{M}_1 = \mathbf{R}_1 \rhd_1 (\mathbf{B}_{1,2} \lhd_1 \mathbf{F}_2)$$

Hence we see that we have in fact used a pair of semimodules with identical additive components: a left semimodule $(N_1, \boxplus_1, \lhd_1)$ over N_2 and a right semimodule $(N_1, \boxplus_1, \rhd_1)$ over S.

Completing the example of Figure 10.1, the routing matrix for G_1 is computed using the semiring MinPlus. The resulting matrix $\mathbf{R}_1$ is shown in Figure 10.1(g). We combine $\mathbf{R}_1$ with the mapping matrix $\mathbf{M}_1$ using the left version of the semimodule

$$\text{Hot}(\text{MinPlus}, \text{Cold}(\text{MaxMin}, \text{Min})).$$

The resulting forwarding table $\mathbf{F}_1$ is given in Figure 10.1(h). The bold arrows in Figure 10.1(a) denote the forwarding paths from node 1 to destinations d_1 and d_2. Note that the two egress nodes (4 and 5) from G_1 are at identical distances from 1, and therefore the bandwidth components from G_2 are used as tie-breakers. This results in a different egress point for each destination.

Future work might involve extending the simple redistribution model of Section 10.2 to take account of the situation where V_1, V_2 are D not pair-wise disjoint. A practical instance of this situation occurs when routing domains overlap. This forces routers to select between multiple routes (each from a different routing protocol) to the same destination. Recall from Chapter 1 that the current mechanism to resolve this ambiguity is *administrative distance*, whereby the protocols are statically 'ranked'. For example, routes from OSPF may be preferred to those from RIP. Algebraically, this may be modelled using an ordered disjoint union, although additional work remains to be done in this area.

10.3 Relation to routing scalability problem

In this section we conclude by relating our simple redistribution model to proposals for creating a more scalable routing and addressing system for the Internet.

10.3.1 IAB problem statement

The Internet Architecture Board (IAB) Routing and Addressing Workshop, which occurred in 2006, was convened to 'develop a shared understanding of the problems that the large backbone operators are facing regarding the scalability of today's Internet routing system' [126]. The main consensus of the workshop was that it is essential to create a 'scalable routing and addressing system, one that is scalable in the face of multihoming, and that facilitates a wide spectrum of traffic engineering (TE) requirements' [126].

Currently, the full BGP routing table results in over 300,000 entries within the forwarding table, and the trend is for this number to keep increasing at a super-linear rate [1]. Increased numbers of routes lead to increased BGP convergence times. They also require upgrades to the amount of FIB memory, which includes the more expensive Content Addressable Memory (CAM) type. The predicted trend is that routers will become increasingly expensive due to the ever-more specialised technology needed to handle the high growth rate of the core routing table [126].

One primary reason for the growth of the core routing table is that many edge networks are requesting the assignment of provider-independent (PI) addresses. Such address blocks are assigned to the edge network themselves, and are therefore not topologically aggregable. This contrasts with the traditional approach whereby edge networks are given provider-assigned (PA) addresses, which are taken from the address blocks of service providers. The main advantage of PI addresses over PA address is that they permit a greater degree of traffic engineering at multi-homed sites. PI addresses also allow edge networks to avoid renumbering when changing providers.

10.3.2 Locator/identifier separation

One popular approach to addressing the Internet scalability problems is to disentangle the notions of host location and host identity. These two concepts are currently intertwined in IP networking, often necessitating the use of PI addressing to allow traffic engineering. The essential idea in a locator/identifier separation scheme is that each end-host has a long-term *identifier* that is independent of any networks that the host might be attached to. Each network interface on a host then has a topologically-significant *locator*, which may also be thought of as a network prefix. Locators remain restricted to the network layer, whilst identifier are used at the transport layer and above.

Locator/identifier separation allow edge-network egress routers the ability to rewrite locators, permitting traffic engineering and also mobility (in some proposals), without resorting to PI addressing [127]. Critically, locator rewriting does not affect transport-level connectivity. This is because locators are restricted to the network layer, and

thus are free to change without affecting upper-layer protocols. Note that one minor disadvantage of locator/identifier separated schemes is that legacy protocols such as FTP might break due to their application-level use of locators.

Specific proposals for locator/identifier separated architectures include Shim6 [128], ILNP [129], as well as the earlier GSE proposal [130]. Shim6 primarily focuses upon load-sharing and failover for multi-homed sites, whereas ILNP is more ambitious in supporting both traffic engineering and mobility. ILNP relies on DNS to map DNS names to identifier and locator values.

We view metarouting as being generally applicable in the context of a locator/identifier separated Internet; such an architecture still necessarily requires routing protocols to compute paths both within the core network and also at edge networks. We might even imagine a situation where the adoption of a locator/identify split allows increased innovation in routing protocol design.

10.3.3 Map/encap

An alternative approach to addressing Internet scalability problems is to *encapsulate* (tunnel) traffic as it passes over the core Internet. In this way, only the addresses of core routers need be present in the routing table. Such schemes, as found in LISP [41] and its variants, often use the terms locator and identifier, although with markedly different meanings from the previous section; locators now refer to the addresses of routers, whilst identifiers are the addresses of edge nodes. Ironically, the pure meaning of locators and identifiers still remain intertwined with this approach. Prior to performing encapsulation, ingress edge routers must consult a *mapping service* to translate from (impure) end-node identifiers to egress router locators. Hence this approach is often given the name *map/encap*.

Recent work suggests that adopting a map/encap scheme for the Internet might lead to a reduction of two orders of magnitude in the number of FIB entries [131]. However, this figure is dependent upon a number of factors, such as a universal adoption map/encap. An additional benefit of several map/encap proposals is that it is only necessary to modify edge routers; end-hosts are unaffected, in contrast to pure locator/identifier schemes. However, a disadvantage of map/encap is that host mobility remains problematic; a roaming host must still change its (impure) identifier, with associated disruptions to transport-level connectivity.

10.3.4 Simple route redistribution as a model of map/encap

The simple route redistribution model can be viewed as a model of a map/encap system. Within our model, destinations D correspond to the names of edge nodes, whilst the

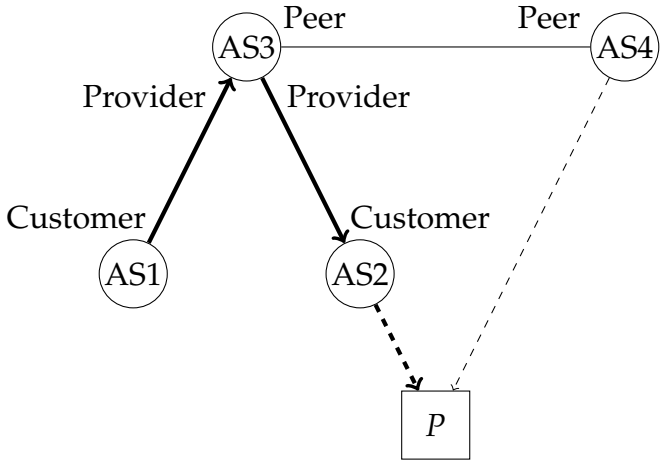


Figure 10.2: Example of AS-level connectivity graph illustrating potential for distributivity violations. AS 3 will normally chose customer route over peer route, irrespective of the preferences of AS 1.

node addresses V_1, V_2 correspond to core router addresses. Routing matrices $\mathbf{R}$ are analogous to core routing tables, containing only node addresses $v \in V_1 \cup V_2$. Mapping matrices $\mathbf{M}$ are abstract versions of mapping services, translating from destinations $d \in D$ to (implicit) egress node addresses $v \in V_1 \cup V_2$.

The fundamental problem that map/encap systems are attempting to solve is that the Internet-wide forwarding table $\mathbf{F}$ is extremely large; currently, it is necessary for a router i to maintain an entire row $\mathbf{F}(i, \_)$ to be able to reach all possible destinations $D \cup V_1 \cup V_2$. With a map/encap scheme, a router may instead store a much small routing table $\mathbf{R}(i, \_)$ to reach nodes $V_1 \cup V_2$. A router may then dynamically construct *individual* entries $\mathbf{F}(i, d)$ using the mapping service $\mathbf{M}$. Indeed, we might expect the number of entries within the mapping service to be far greater than the number contained within the global routing table.

We now consider the case in which a non-distributive routing protocol is used to construct forwarding tables. In such cases, we show how adopting a map/encap architecture can allow originating networks to bypass restrictions imposed by non-distributivity. One particular outcome is that originating networks may select paths which are globally optimal, instead of just locally optimal. Note that BGP is a non-distributive protocol [91], and therefore this section is relevant to the current Internet. We first consider the case in which there is no map/encap architecture, as found in the current Internet.

Suppose that there is the situation illustrated in Figure 10.2, where there are four autonomous systems (ASes) on the Internet, each of which is running BGP for inter-AS connectivity. AS 1 and AS 2 are both customers of AS 3, whilst AS 4 is a peer of AS 3. We assume that routes from customers are strictly preferred to those from peers, which are in turn strictly preferred to those from providers. AS 2 and AS 4 both advertise the

same prefix P. This could occur if P is originated by a multi-homed network attached to both ASes. AS 3 prefers routes from AS 2 (a customer) over those from AS 4 (a peer). Now suppose that AS 1 instead prefers routes from AS 4 over those from AS 2. In the simplest case, this could be because AS 2 artificially pads the AS path attribute of routes. This inversion of route preferences between AS 3 and AS 1 is a violation of distributivity.

On the current Internet, this non-distributivity causes AS 1 to chose AS 2's route to P because AS 3 never propagates the route from AS 4. The result is that AS 1 chooses a path to P that is only locally optimal instead of globally optimal; AS 1 chooses the best path to P *amongst all those that is has been given*, instead of amongst all possible paths. From the point of view of the originator of P, this non-distributivity means that it is very difficult to control incoming traffic; changes in the the policy of the originating AS are not respected by AS 3. The only way to force AS 1 to use the path via AS 4 is to specially configure community attributes on P to affect the route decision process at AS 3.

Now consider the case where the Internet is running BGP with a map/encap architecture. BGP is used to compute shortest paths between core nodes. The attachment points of edge nodes are determined using a mapping service e.g. the mapping service would inform AS 1 that P is originated by routers within both AS 2 and AS 4. AS 1 can now reach P by tunnelling to an egress point in AS 4, the source of closest originating locator for P. i.e. *AS 1 is no longer constrained by AS 3's preferences for AS 2 over AS 4*. From the point of view of the originator of P, it is now much easier to use local policy to control incoming traffic.

In practice, the choice of egress point could be determined by both the routing distance and also policy expressed within the mapping service itself. Furthermore, the mapping service policy could be a function of the source locator e.g. the mapping service could return different responses for routers within AS 1 and AS 3.

Chapter 11

Conclusions

In this chapter we summarise the research results presented in this dissertation and show how they support the thesis (§ 11.1). We then conclude with an outline of areas future work (§ 11.2).

11.1 Summary

Our thesis from Chapter 1 stated that:

> Using the theory of algebraic routing, routing protocols can be specified at a high-level, automatically checked for correctness and then compiled into efficient implementations. The resulting protocols are easier to understand and require significantly less implementation effort.

We now summarise the research presented in this dissertation and evaluate the extent to which it supports the thesis.

We commenced this dissertation by showing how it is difficult both to understand and implement *existing* routing protocols (§ 1). This sets the benchmark for substantiating the statement that our generated protocols are *'easier to understand and require significantly less implementation effort'*. We demonstrated that due to the cost of creating new routing protocols, network operators are instead 'making do' with existing routing protocols. We also described limitations of the current mechanisms for protocol interaction. Next we outlined several instances in which current routing protocols behave in apparently unpredictable manners due to the absence of a clear semantics. Finally, we described examples of errors that have occurred in current routing protocol implementations.

We then presented the *'theory of algebraic routing'* and described how it can be used to separate routing languages from routing algorithms (§ 2). We also described how the

algebraic properties of routing languages can be *'automatically check[ed] for correctness'* to guarantee the safety of generated routing protocols. Finally, we exhibited several other examples of systems in which domain-specific languages have been applied to networking problems for the purpose of increasing clarity of specification and aiding implementation. We showed how the approach of compiling routing languages from high-level specifications can be viewed as an instance of this approach.

We now discuss the metarouting architecture for compiling routing protocol specifications. In Chapter 3 we described how the compilation task can be decomposed into a number of stages. We also showed how current routing protocols can be generalised to produce online routing algorithms. This in itself dramatically reduces implementation effort. In Chapter 6, we discussed the specifics of compilation. Our approach involves instantiating a library of templated C++ code. We showed how this technique can be viewed as using the C++ template facilities to write a domain-specific embedded language. The benefits of this approach are that the compiler itself generates minimal code and that the templated code can be specialised for increased performance.

We have demonstrated two metalanguages that allow routing languages to be *'specified at a high level'*. In Chapter 5, we demonstrated our first metalanguage, RAML_1. This language has an associated semantics and intermediate language, which is described in Chapter 4. Compared to BGP, which does not have a formal semantics, this facility means that routing protocols generated using metarouting are indeed *'easier to understand'*. We then extended the metalanguage in Chapter 7 in order to increase its expressivity. We motivated this extension using the 'regions' and 'minimal paths' examples. Both of these examples could be of practical use. They also demonstrate that protocols can indeed be developed with *'significantly less effort'*. Note that whether *all* useful routing languages are expressible within our metalanguage is impossible to answer in the positive; we return to this question in Section 11.2.

We now examine whether our routing protocol specifications are compiled into *'efficient implementations'*. In Chapter 8 we evaluated the performance of three example routing languages, and demonstrated how to improve their efficiency. We demonstrated how to systematically exploit sharing, an optimisation technique that is manually performed in current routing protocols. Using memoisation, we showed that in some instances it is possible to trade-off memory usage for execution time. We also illustrated an example of using automatically derived algebraic properties to increase the efficiency of the generated code. Whilst we lack any alternatives to which we might compare the efficiency of our generated code, we have certainly demonstrated techniques that allow us to improve the efficiency of code that our system itself generates.

Turning to Chapter 9, we discussed the differences between routing and forwarding. We demonstrated how semimodules could be used to model attachment, and demonstrated the *hot-potato* and *cold-potato* forwarding methods. Both of these forwarding methods

are found in use in inter-domain routing on the Internet. Finally, in Chapter 10 we described how to extend the attachment model to represent a restricted form of route redistribution that we term *simple* route redistribution. This is the first algebraic account of redistribution, and represents a first step in formally understanding the 'glue logic' of the Internet. Whilst we have not yet incorporated the routing/forwarding or the simple route redistribution models into the metarouting system, they represent additional dimensions in which routing protocols might be specified so that they are *'easier to understand'*.

11.2 Future work

Whilst we have demonstrated the feasibility of the metarouting approach, much research still remains to be done within this area. We now discuss areas in which the work in this dissertation might be extended.

From a practical point of view, there are several areas of work remaining in order to produce robust routing protocols that are suitable for operational use. It would be desirable to support a greater range of online routing algorithms. For example, we would like to include routing algorithms from the XORP routing platform [22]. Supporting both Quagga and XORP would help demonstrate the generality of our approach. We would also like to add support for Mobile Ad hoc Networking (MANET) algorithms, such as that employed by the Ad hoc On-Demand Distance Vector (AODV) routing protocol [132]. It is also necessary to understand how to permit the flexible configuration of routing protocols in a distributed environment, whilst still maintaining safety. For example, we might wish to permit different components of policy to be specified at each end of routing adjacencies. Some promising initial work into this area has already been conducted [114].

Another area of research concerns the optimisation of generated code. Whilst we have shown that general techniques such as sharing can be successfully applied, we have used relatively few of the rich set of algebraic properties that are automatically determined at compile time. Other research [121, 122] suggests that these properties may be used to further optimise minimal sets constructions. A related area concerns 'tunable' code generation. Memoisation demonstrates that it is possible to reduce the execution time of (offline) routing protocols at the expense of greater memory use. We would like to classify optimisations so that they can be used to target platforms with a range of resources. For example, the memoisation technique might be more appropriate for a device with a large amount of memory, whilst the sharing optimisation might be particularly suited to a system in with little memory. As a further extension, we would also like to discover to what extent the particular choice of routing algorithm affects the performance of the routing language.

Turning to the design of the compiler, it would be desirable to automatically generate the property-checking rules. These rules are used within the compiler for verifying the correctness of routing language specifications. Our current approach is to prove these rules 'on paper', and then manually implement the corresponding property-checking code. The main problem is that it is necessary to track dozens of different properties and rules, causing the compiler implementation to become substantially more complex. Automated theorem proving might be used to address this issue. We envision a situation in property rules are formally specified and checked. Property-checking code for the compiler could then be automatically extracted from formal specifications.

There also remain areas of work in the design of the metalanguage. We need to understand whether our metalanguage is sufficiently expressive; we hope that users of our system will not be needlessly constrained by the current selection of language operators. We believe that we can answer this question in part by having network operators and researchers use our system within a practical context. Knowledge gained from this experience may well feed back into the design of further extensions for the metalanguage.

We would like to extend the metarouting system to incorporate the attachment models from Chapter 9. Part of this work would involve extending the metalanguage to support the hot-potato and cold-potato semimodules, for example. We would also like to further modularise online algorithms to support separate mapping services. With regards to simple route redistribution, it would seem desirable to have the ability to generate first-class redistribution protocols that operate above the routing layer. We have also not fully addressed issues such as mutual redistribution or administrative distance.

What is the future of metarouting? Currently, network operators 'make do' with a small selection of monolithic routing protocols, and network researchers struggle to implement their ideas. Metarouting has the potential to revolutionise this situation by allowing the rapid development of new, provably-correct routing protocols.

Bibliography

[1] G. Huston. AS6447 BGP Routing Table Analysis Report. `http://bgp.potaroo.net/as6447/`, 2009. 1, 1.2.1, 10.3.1

[2] P. Hoffman and S. Harris. RFC 4677: The Tao of IETF: A Novice's Guide to the Internet Engineering Task Force, September 2006. 1

[3] Timothy G. Griffin and João Luís Sobrinho. Metarouting. *SIGCOMM Comput. Commun. Rev.*, 35(4):1–12, 2005. 1, 1.6, 7.2

[4] Bernard Carré. *Graphs and Networks*. Oxford University Press, Oxford, 1979. 1, 2.6.3, 2.6.6, 7.2

[5] M. Gondran and M. Minoux. *Graphs, Dioids, and Semirings : New Models and Algorithms*. Springer, 2008. 1, 2.7.3, 7.2

[6] V. Fuller and T. Li. RFC 4632: Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan, August 2006. 1.1.1

[7] G. Huston. CIDR Report. `http://www.cidr-report.org/as2.0/`, 2009. 1.1.2

[8] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4 (BGP-4), March 1995. 1.1.2, 1.1.4, 2.1

[9] Y. Rekhter, T. Li, and S. Hares. RFC 4271: A Border Gateway Protocol 4 (BGP-4), January 2006. 1.1.2, 1.5, 2.7.1, 9.3.4

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 1.1.3, 2.1, 2.7.2

[11] J. Moy. RFC 2328: OSPF Version 2, April 1998. 1.1.3, 2.1, 2.7.2, 9.4

[12] D. Oran (Ed.). RFC 1142: OSI IS-IS Intra-domain Routing Protocol, February 1990. 1.1.3, 2.1, 2.7.2

[13] C. Hedrick. RFC 1058: Routing Information Protocol, June 1998. 1.1.3, 2.7.1

[14] 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. 1.1.3

[15] J. J. Garcia-Luna-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Trans. Netw.*, 1(1):130–141, 1993. 1.1.3

[16] Iljitsch van Beijnum. *BGP: Building Reliable Networks with the Border Gateway Protocol*. O'Reilly and Associates, Inc., 2002. 1.1.4

[17] Alexander Gurney and Timothy G. Griffin. Lexicographic products in metarouting. In *Proc. ICNP 2007*, pages 113–122, October 2007. 1.1.4, 1.6, 4.3.2, 4.4.3

[18] G. Huston. Interconnection, Peering, and Settlements. `http://www.potaroo.net/papers/1999-6-peer/peering.pdf`, 1999. 1.1.5

[19] Russ White, Danny McPherson, and Srihari Sangli. *Practical BGP*. Addison-Wesley, 2005. 1.2.1

[20] Randy Zhang and Micah Bartell. *BGP Design and Implementation*. Cisco Press, December 2003. 1.2.1

[21] Nick Feamster, Jay Borkenhagen, and Jennifer Rexford. Guidelines for Interdomain Traffic Engineering. *SIGCOMM Comput. Commun. Rev.*, 33(5):19–30, 2003. 1.2.1

[22] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible IP router software. In *Proc. NSDI '05*, Boston, MA, USA, May 2005. 1.2.2, 1.5, 2.3.2, 3.4, 11.2

[23] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Morley Mao, Scott Shenker, and Ion Stoica. HLP: a next generation interdomain routing protocol. In *Proc. SIGCOMM '05*, pages 13–24, New York, NY, USA, 2005. ACM. 1.2.2

[24] Franck Le, Geoffrey G. Xie, Dan Pei, Jia Wang, and Hui Zhang. Shedding light on the glue logic of the internet routing architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):39–50, 2008. 1.3, 10.1

[25] David A. Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtýsson, and Albert Greenberg. Routing design in operational networks: a look from the inside. *SIGCOMM Comput. Commun. Rev.*, 34(4):27–40, 2004. 1.3

[26] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks. In *Proc. INFOCOM '05*, 2005. 1.3

[27] Franck Le and Geoffrey G. Xie. On guidelines for safe route redistributions. In *Proc. INM '07*, pages 274–279, New York, NY, USA, 2007. ACM. 1.3, 10.1

[28] Franck Le, Geoffrey G. Xie, and Hui Zhang. Understanding route redistribution. In *Proc. ICNP '07*, pages 81–92. IEEE, October 2007. 1.3, 10.1

[29] Franck Le, Geoffrey G. Xie, and Hui Zhang. Instability Free Routing: Beyond One Protocol Instance. In *Proc CONEXT '08*, pages 1–12, New York, NY, USA, 2008. ACM. 1.3, 10.1

[30] Timothy G. Griffin and Geoff Huston. RFC 4264: BGP Wedgies, November 2005. 1.4.1

[31] Timothy G. Griffin and Gordon Wilfong. Analysis of the MED Oscillation Problem in BGP. *Proc. ICNP '02*, pages 90–99, 2002. 1.4.2, 1.4.2

[32] Cisco. Field Notice: Endless BGP Convergence Problem in Cisco IOS Software Releases. `http://www.cisco.com/en/US/ts/fn/100/fn12942.html`, October 2000. 1.4.2

[33] Timothy G. Griffin and Gordon Wilfong. An Analysis of BGP Convergence Properties. In *Proc. SIGCOMM '99*, pages 277–288, New York, NY, USA, 1999. ACM. 1.4.2

[34] Kannan Varadhan, Ramesh Govindan, and Deborah Estrin. Persistent Route Oscillations in Inter-Domain Routing. *Computer Networks*, 32(1):1–16, 2000. 1.4.2

[35] D. McPherson, V. Gill, D. Walton, and A. Retana. RFC 3345: Border Gateway Protocol (BGP) Persistent Route Oscillation Condition, August 2002. 1.4.2

[36] Quagga routing suite. `http://www.quagga.net/`. 1.5, 1.6, 2.3.1, 3.4

[37] Earl Zmijewski. Longer is not always better. `http://www.renesys.com/blog/2009/02/longer-is-not-better.shtml`, February 2009. 1.5

[38] Earl Zmijewski. Reckless Driving on the Internet. `http://www.renesys.com/blog/2009/02/the-flap-heard-around-the-worl.shtml`, February 2009. 1.5

[39] Earl Zmijewski. AfNOG Takes Byte Out of Internet. `http://www.renesys.com/blog/2009/05/byte-me.shtml`, May 2009. 1.5

[40] OpenBGPD. `http://www.openbgpd.org/`. 1.5

[41] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Internet Draft: Locator/ID Separation Protocol (LISP), July 2009. 1.6, 10.3.3

[42] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. A brief history of the internet. `http://www.isoc.org/internet/history/brief.shtml`, December 2003. 2.1

[43] J. C. R. Licklider. Memorandum For Members and Affiliates of the Intergalactic Computer Network. `http://www.kurzweilai.net/articles/art0366.html`, April 1963. 2.1

[44] Leonard Kleinrock. Information Flow in Large Communication Nets, Proposal for a Ph.D. Thesis. `http://www.cs.ucla.edu/~lk/LK/Bib/REPORT/PhD/`, May 1961. 2.1

[45] John McQuillan and David Walden. The ARPA Network Design Decisions. *Computer Networks*, 1(5):243–289, August 1977. 2.1

[46] Vinton G. Cerf and Robert E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE/ACM Trans. Comm.*, 22(5), May 1974. 2.1

[47] Vinton G. Cerf, Yogen Dalal, and Carl Sunshine. RFC 675: Specification of Internet Transmission Control Program, December 1974. 2.1

[48] Jon Postel (Ed.). RFC 760: Internet Protocol, January 1980. 2.1

[49] Jon Postel (Ed.). RFC 793: Transmission Control Protocol, September 1981. 2.1

[50] John M. McQuillan, Ira Richer, and Eric C. Rosen. The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications*, 28(5), May 1980. 2.1

[51] J. Postel. RFC 801: NCP/TCP transition plan, November 1981. 2.1

[52] Robert Hinden and Alan Sheltzer. RFC 823: The DARPA Internet Gateway, September 1982. 2.1

[53] R. Braden and J. Postel. RFC 1009: Requirements for Internet Gateways, June 1987. 2.1

[54] Eric C. Rosen. RFC 827: Exterior Gateway Protocol (EGP), October 1982. 2.1

[55] Shivkumar Kalyanaraman. Exterior Gateway Protocols: EGP, BGP-4, CIDR. `http://www.ecse.rpi.edu/Homepages/shivkuma/teaching/sp2000/i12_egp/index.htm`, 2000. 2.1

[56] L. Landweber, M. Litzkow, D. Neuhengen, and M. Soloman. Architecture of the CSNET name server. In *Proc. COMM '83*, pages 146–153. ACM, 1983. 2.1

[57] J. Rekhter. RFC 1092: EGP and Policy Based Routing in the New NSFNET Backbone, Februrary 1989. 2.1

[58] H. W. Braun. RFC 1093: The NSFNET Routing Architecture, February 1989. 2.1

[59] K. Lougheed and Y. Rekhter. RFC 1105: A Border Gateway Protocol (BGP), June 1989. 2.1

[60] K. Lougheed and Y. Rekhter. RFC 1163: Border Gateway Protocol (BGP), June 1990. 2.1

[61] A Brief History of NSF and the Internet. `http://www.nsf.gov/news/news_summ.jsp?cntn_id=103050`, 2003. 2.1

[62] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. 2.2.1

[63] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering – Functional Pearl. In *Proc. ICFP '00*, Montreal, Canada, September 2000. 2.2.1

[64] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. `http://dinosaur.compilertools.net/yacc/`, 1975. 2.2.1, 6.2.1

[65] Stuart I. Feldman. Make – A Program for Maintaining Computer Programs. *Softw., Pract. Exper.*, 9(4):255–265, 1979. 2.2.1

[66] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Professional, second edition, 1994. 2.2.1

[67] Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28, 1996. 2.2.1

[68] Peter J. McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. *SIGCOMM Comput. Commun. Rev.*, 30(4):321–333, 2000. 2.2.3

[69] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. BinPAC: a YACC for writing application protocol parsers. In *Proc. IMC '06*, pages 289–300, New York, NY, USA, 2006. ACM. 2.2.3

[70] Godmar Back. DataScript – A Specification and Scripting Language for Binary Data. *Lecture Notes in Computer Science*, 2487/2002:66–77, 2002. 2.2.3

[71] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proc. PLDI '05*, June 2005. 2.2.3

[72] Intel Corporation. Intel IXP2400 Network Processor: Flexible, High-Performance Solution for Access and Edge Applications. `http://download.intel.com/design/network/papers/ixp2400.pdf`, 2002. 2.2.4

[73] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear Types for Packet Processing. In *ESOP '04: Proceedings of the 13th European Symposium on Programming*, pages 204–218. Springer, 2004. 2.2.4

[74] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *Proc. SIGMOD '06*, June 2006. 2.2.5

[75] James Kelly, Wladimir Araujo, and Kallol Banerjee. Rapid Service Creation using the JUNOS SDK. In *Proc. PRESTO '09*. ACM, August 2009. 2.3.3

[76] Extreme Networks. ExtremeXOS Operating System, Version 12.3. `www.extremenetworks.com/libraries/products/DSExtXOS_1030.pdf`, 2009. 2.3.3

[77] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000. 2.4.1

[78] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA – An Open Platform for Gigabit-rate Network Switching and Routing. In *Proc. MSE '07*, June 2007. 2.4.2

[79] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. `http://www.openflowswitch.org//documents/openflow-wp-latest.pdf`, March 2008. 2.4.3

[80] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998. 2.5.1

[81] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert L. Constable. Building Reliable, High-Performance Systems from Components. *Operating Systems Review*, 34(5):80–92, 1999. 2.5.1

[82] Edoardo S. Biagioni. A Structured TCP in Standard ML. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994. 2.5.2

[83] Edoardo Biagioni, Robert Harper, and Peter Lee. A network protocol stack in Standard ML. *Higher-order and symbolic computation*, 14(4):309–356, December 2001. 2.5.2

[84] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. The MIT Press, second edition, May 1997. 2.5.2

[85] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A Readable TCP in the Prolac Protocol Language. In *Proc. SIGCOMM '99*, pages 3–13, Cambridge, Massachusetts, USA, August 1999. 2.5.3

[86] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. Melange: creating a 'functional' internet. In *Proc. EuroSys '07*, pages 101–114, New York, NY, USA, 2007. ACM. 2.5.4

[87] R. C. Backhouse and B. A. Carré. Regular Algebra Applied to Path-finding Problems. *J. Inst. Math. Appl.*, 15(2), 1975. 2.6.3

[88] M. Gondran and M. Minoux. *Graphs and Algorithms*. John Wiley and Sons, 1984. 2.6.3, 7.2

[89] John M. Howie. *Fundamentals of Semigroup Theory*. Oxford University Press, USA, February 1996. 2.6.3

[90] João Luís Sobrinho. Network routing with path vector protocols: theory and applications. In *Proc. SIGCOMM '03*, pages 49–60, New York, NY, USA, 2003. ACM. 2.6.7

[91] Timothy G. Griffin and Alexander J. T. Gurney. *Increasing Bisemigroups and Algebraic Routing*, volume 4988/2008. Springer Berlin / Heidelberg, 2008. 2.6.7, 7.2, 7.4, 9.5, 10.3.4

[92] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1), 1958. 2.7.1

[93] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001. 2.7.1, 2.7.2

[94] H. N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31:148–168, 1985. 2.7.2

[95] Mikkel Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. ACM*, 46(3):362–394, 1999. 2.7.2

[96] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. 4.2.2

[97] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.11. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, November 2008. 4.5.1

[98] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, February 1993. 4.5.3

[99] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, February 2002. 4.5.3, 5.2

[100] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. 6.1.1

[101] Todd L. Veldhuizen. C++ Templates as Partial Evaluation. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation '99*, Tech. Report NS-99-1, pages 13–18. BRICS, 1999. 6.1.1

[102] Eigen C++ template library for linear algebra. `http://eigen.tuxfamily.org`. 6.1.1

[103] Blitz++. `http://www.oonumerics.org/blitz/`. 6.1.1

[104] Peter Gottschling, David S. Wise, and Michael D. Adams. Representation-transparent matrix algorithms with scalable performance. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125, New York, NY, USA, 2007. ACM. 6.1.1

[105] GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`. 6.2.1

[106] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, February 2001. 6.2.3

[107] T. Lengauer and D. Theune. Efficient algorithms for path problems with general cost criteria. In *Proceedings of the 18th international colloquium on Automata, languages and programming*, pages 314–326, New York, NY, USA, 1991. Springer-Verlag New York, Inc. 7.2

[108] T. Lengauer and D. Theune. Unstructured path problems and the making of semirings. *Lecture Notes in Computer Science*, 519/1991, 1991. 7.2

[109] L. Fuchs. *Partially Ordered Algebraic Systems*. Pergamon Press, 1963. 7.2

[110] G. Birkhoff. *Lattice Theory*. American Mathematical Society, second edition, 1967. 7.2

[111] R. E. Johnson. Free products of ordered semigroups. *Proceedings of the American Mathematical Society*, 19(3):697–700, 1968. 7.2

[112] João Luís Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet. *IEEE/ACM Trans. Netw.*, 10(4):541–550, 2002. 7.2

[113] João Luís Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Trans. Netw.*, 13(5):1160–1173, 2005. 7.2

[114] Philip J. Taylor and Timothy G. Griffin. A model of configuration languages for routing protocols. In *Proc. PRESTO '09*, 2009. 7.2.2, 11.2

[115] E. Minieka and D. R. Shier. A Note on an Algebra for the k Best Routes in a Network. *J. Inst. Maths Applics*, 11:145–149, August 1972. 7.2.3

[116] Jean-Christophe Filliâtre and Sylvain Conchon. Type-Safe Modular Hash-Consing. In *Proc. ACM SIGPLAN Workshop on ML '06*, Portland, Oregon, September 2006. 8.1.1, 8.1.3

[117] John Allen. *Anatomy of LISP*. McGraw-Hill, 1978. 8.1.1

[118] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994. 8.1.1

[119] Donald R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968. 8.1.3

[120] Chris Okasaki and Andrew Gill. Fast Mergeable Integer Maps. In *Proc. ACM SIGPLAN Workshop on ML '98*, pages 77–86, Baltimore, Maryland, USA, 1998. 8.2, 8.1.3

[121] Alberto Martelli. An Application of Regular Algebra to the Enumeration of Cut Sets in a Graph. In *IFIP Congress*, pages 511–515, 1974. 8.2.1, 8.4.1, 11.2

[122] Alberto Martelli. A Gaussian Elimination Algorithm for the Enumeration of Cut Sets in a Graph. *J. ACM*, 23(1):58–73, 1976. 8.2.1, 8.4.1, 11.2

[123] Béla Bollobás. *Random Graphs*. Cambridge University Press, second edition, January 2001. 8.2.3

[124] John N. Billings and Timothy G. Griffin. A model of Internet routing using semi-modules. In *Relations and Kleene Algebra in Computer Science*. Springer Berlin / Heidelberg, November 2009. 9

[125] John T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1998. 9.4

[126] D. Meyer, L. Zhang, and K. Fall. RFC 4984: Report from the IAB Workshop on Routing and Addressing, September 2007. 10.3.1

[127] Geoff Huston. RFC 4177: Architectural Approaches to Multi-homing for IPv6, September 2005. 10.3.2

[128] E. Nordmark and M. Bagnulo. RFC 5533: Shim6: Level 3 Multihoming Shim Protocol for IPv6, June 2009. 10.3.2

[129] Randall Atkinson, Saleem Bhatti, and Stephen Hailes. ILNP: mobility, multihoming, localised addressing and security through naming. *Telecommunication Systems*, 42(3–4):273–291, December 2009. 10.3.2

[130] Mike O'Dell. Internet Draft: GSE - An Alternate Addressing Architecture for IPv6 , February 1997. 10.3.2

[131] B. Quoitin, L. Iannone, C. de Launois, and O. Bonaventure. Evaluating the Benefits of the Locator/Identifier Separation. In *Proc. MobiArch '07*, August 2007. 10.3.3

[132] C. Perkins, E. Belding-Royer, and S. Das. RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing, July 2003. 11.2